Natural Language Understanding with Distributed Representation

Kyunghyun Cho
New York University

November 22, 2015

This is a lecture note used for the course DS-GA 3001 ⟨Natural Language Understanding with Distributed Representation⟩ at the Center for Data Science[1], New York University in Fall, 2015. This lecture note will be constantly updated throughout the semester. Please, check for the latest version frequently.

[1] http://cds.nyu.edu/

Contents

1 Introduction **5**
1.1 Route we will *not* take 5
1.1.1 What is Language? 5
1.1.2 Language Understanding 6
1.2 Road we *will* take 8
1.2.1 Language as a Function 8
1.2.2 Language Understanding as a Function Approximation 8

2 Function Approximation as Supervised Learning **11**
2.1 Function Approximation: Parametric Approach 11
2.1.1 Expected Cost Function 11
2.1.2 Empirical Cost Function 12
2.2 Learning as Optimization 13
2.2.1 Gradient-based Local Iterative Optimization 13
2.2.2 Stochastic Gradient Descent 14
2.3 When do we stop learning? 16
2.3.1 Early Stopping 16
2.3.2 Model Selection 18
2.4 Evaluation 19
2.5 Linear Regression for Non-Linear Functions 20
2.5.1 Feature Extraction 20

3 Neural Networks and Backpropagation Algorithm **22**
3.1 Conditional Distribution Approximation 22
3.1.1 Why do we want to do this? 24
3.1.2 Other Distributions 25
3.2 Feature Extraction is also a Function 25
3.3 Multilayer Perceptron 26
3.3.1 Example: Binary classification with a single hidden unit . . . 27
3.3.2 Example: Binary classification with more than one hidden units 29
3.4 Automating Backpropagation 31
3.4.1 What if a Function is *not* Differentiable? 32

4 Recurrent Neural Networks and Gated Recurrent Units **35**
4.1 Recurrent Neural Networks 35
4.1.1 Fixed-Size Output y 37
4.1.2 Multiple Child Nodes and Derivatives 38
4.1.3 Example: Sentiment Analysis 39
4.1.4 Variable-Length Output y: $|x| = |y|$ 40
4.2 Gated Recurrent Units 43
4.2.1 Making Simple Recurrent Neural Networks *Realistic* 43
4.2.2 Gated Recurrent Units 44
4.2.3 Long Short-Term Memory 46
4.3 Why not Rectifiers? 47
4.3.1 Rectifiers Explode 47
4.3.2 Is tanh a Blessing? 49
4.3.3 Are We Doomed? 52
4.3.4 Gated Recurrent Units Address Vanishing Gradient 53

5 Neural Language Models **55**
5.1 Language Modeling: First Step 55
5.1.1 What if those linguistic structures do exist 56
5.1.2 Quick Note on Linguistic Units 57
5.2 Statistical Language Modeling 58
5.2.1 Data Sparsity/Scarcity 59
5.3 n-Gram Language Model 61
5.3.1 Smoothing and Back-Off 62
5.3.2 Lack of Generalization 65
5.4 Neural Language Model 66
5.4.1 How does Neural Language Model Generalize to Unseen n-Grams? – Distributional Hypothesis 68
5.4.2 Continuous Bag-of-Words Language Model: Maximum Pseudo–Likelihood Approach 71
5.4.3 Semi-Supervised Learning with Pretrained Word Embeddings 74
5.5 Recurrent Language Model 76
5.6 How do n-gram language model, neural language model and RNN-LM compare? 79

6 Neural Machine Translation **82**
6.1 Statistical Approach to Machine Translation 82
6.1.1 Parallel Corpora: Training Data for Machine Translation . . . 84
6.1.2 Automatic Evaluation Metric 87
6.2 Neural Machine Translation: Simple Encoder-Decoder Model 90
6.2.1 Sampling vs. Decoding 91
6.3 Attention-based Neural Machine Translation 97
6.3.1 What does the Attention Mechanism do? 101
6.4 Warren Weaver's Memorandum 103

7 Final Words:
Beyond Natural Languages **107**
7.1 Multimedia Description Generation as Translation 107
7.2 Natural Language Understanding with Word Knowledge 107
7.3 Summary 107

Chapter 1

Introduction

This lecture is going to be the only one where I discuss some philosophical, meaning nonpractical, arguments, because according to Chris Manning and Hinrich Schuetze, "*even practically-minded people have to confront the issue of what prior knowledge to try to build into their model*" [67].

1.1 Route we will *not* take

1.1.1 What is Language?

The very first question we must ask ourselves before starting this course is the question of what natural language is. Of course, the rest of this course does not in any way require us to know what natural language is, but it is a philosophical question I recommend everyone, including myself, to ponder upon once a while.

When I start talking about languages with anyone, there is a single person who never misses to be mentioned, that is Noam Chomsky. His view has greatly influenced the modern linguistics, and although many linguists I have talked to claim that their work and field have long moved on from Chomsky's, I can feel his shadow all over them.

My first encounter with Chomsky was at the classroom of <Automata> from my early undergrad years. I was not the most attentive student back then, and all I can remember is Chomsky's hierarchy and how it has shaped our view on languages, in this context, programming/computer languages. A large part of the course was dedicated to explaining which class of languages emerges given a set of constraints on a set of *generating rules*, or production rules.

For instance, if we are given a set of generating rules that do not depend on the context/meaning of non-terminal symbols (context-free grammar, CFG), we get a context-free language. If we put a bit of constraints to CFG that each generating rule is such that a non-terminal symbol is replaced by either a terminal symbol, a terminal symbol by a non-terminal symbol or an empty symbol, then we get a regular grammar. Similarly to CFG, we get a regular language from the regular grammar, and the regular language is a subset of the context-free language.

What Chomsky believes is that this kind of approach applies also to human languages, or natural languages. There exists a set of generating rules that *generates* a natural language. But, then, the obvious question to follow is where those generating rules are. Where are they stored? How are they stored? Do we have separate generating rules for different languages?

1.1.2 Language Understanding

Understanding Human Language Those questions are interesting, but out of scope for this course. Those questions are the ones linguists try to answer. Generative linguistics aims at figuring out what those rules are, how they are combined to form a valid sentence, how they are adapted to different languages and so on. We will leave these to linguists and continue on to our journey of *building a machine that understands human languages*.

Natural Language Understanding So, let's put these questions aside and trust Chomsky that we, humans, are specially designed to store those generating rules somewhere in the brain [27, 19]. Or, better yet, let's trust Chomsky that there's a universal grammar *built in* our brain. In other words, let's say we were born with this set of generating rules for natural languages, and while growing, we have adapted this universal grammar toward our native tongue (language variation).

When we decide to speak of something (whatever that is and however implausible that is), our brain quickly picks up a sequence of some of those generating rules and starts generating a sentence accordingly. Of course, those rules do not generate a sentence directly, but generates a sequence of control signals to move our muscles to make sound. When heard by other people who understand your language, the sound becomes a sentence.

In our case, we are more interested in a *machine* hearing that sound, or a sentence from here on. When a machine heard this sentence, what would/should a *language understanding machine* do to understand a language, or more simply a sentence? Again, we are assuming that this sentence was generated from applying a sequence of the existing generating rules.

Under our assumption, a natural first step that comes to my mind is to figure out that sequence of the generating rules which led to the sentence. Once the sequence is found, or in a fancier term, inferred, the next step will be to figure out what kind of mental state of the speaker led to those generating rules.

Let's take an example sentence "*Our company is training workers*" (from Sec. 1.3 of [67]), which is a horrible choice, because this was used as an example of ambiguity in parsing. Regardless, a speaker obviously has an awesome image of her company which trains its workers and wants to tell a machine about this. This mental state is used to select the following generating rules (assuming a phrase structure grammar)[1]:

```
(ROOT
```

[1] Stanford Parser: `http://nlp.stanford.edu:8080/parser`

```
(S
    (NP (PRP$ Our) (NN company))
        (VP (VBZ is)
              (VP (VBG training)
                      (NP (NNS workers))))))
```

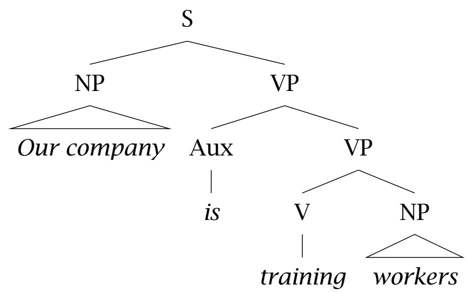


Figure 1.1: A parse of "*Our company is training workers*"

The machine hears the sentence "*Our company is training workers*" and infers the parse in Fig. 1.1. Then, we can make a simple set of rules (again!) to let the machine answer questions about this sentence, kinds of questions that imply that the machine has understood the sentence (language). For instance, given a question "*Who is training workers?*", the machine can answer by noticing that the question is asking for the subject of the verb phrase "*is training*" acted on the object "*workers*" and that the subject is "*Our company*".

Side Note: Bayesian Language Understanding This generative view of languages fits quite well with Bayesian modelling (see, e.g., [73].) There exists a hidden mechanism, or a set of generating rules and a rule governing their composition, which can be modelled as a latent variable Z. Given these rules, a language or a sentence X is generated according to the conditional distribution $P(X|Z)$. Then, understanding language (by humans) is equivalent to computing the posterior distribution over all possible sets of generating rules and their compositional rules (i.e., $P(Z|X)$.) This answers the question of what is the most likely mechanism underlying the observed language.

Furthermore, from the perspective of machines, Bayesian approach is attractive. In this case, we assume to know *the* set of rules in advance and let the latent variable Z denote the specific configuration (use) of those rules. Given this sequence of applying the rules, a sentence X is generated via the conditional distribution $P(X|Z)$. Machine understanding of language is equivalent to inferring the posterior distribution over Z given X.

For more details about Bayesian approaches (in the context of machine learning), please, refer to [12] or take the course DS-GA 1005 Inference and Representation by Prof. David Sontag.

Understanding vs. Using What's clear from this example is that in this generative view of languages, there is a clear separation between understanding and using. Inferring the generating rules from a given sentence is *understanding*, and answering a question based on this understanding, *using*, is a separate activity. Understanding part is done when the underlying (true) structure has been determined regardless of how this understanding be used.

To put it in a slightly different wording, language understanding does not require its use, or downstream tasks. In this road that we will *not* take in this course, understanding exists as it is, regardless of what the understood insight/knowledge will be used for. And, this is the reason why we do not walk down this road.

1.2 Road we *will* take

1.2.1 Language as a Function

In this course, we will view a natural/human language as "*a system intended to communicate ideas from a speaker to a hearer*" [92]. What this means is that we do not view a language as a separate entity that exists on its own. Rather, we view a whole system or behaviour of *communication* as a language. Furthermore, this view dictates that we must take into account the world surrounding a speaker and a hearer in order to understand language.

Under this view of language, language or rather its usage become somewhat similar to action or behaviour. Speaking of something is equivalent to acting on a listener, as both of them influence the listener in one way or another. The purpose of language is then to influence another by efficiently communicate one's will or intention.[2] This hints at how language came to be (or may have come to be): (evolution) language has evolved to facilitate the exchange of ideas among people (learning) humans learn language by being either encouraged or punished for the use of language. This latter view on how language came to be is similar in spirit to the behaviourism of B. F. Skinner ("*necessary mediation of reinforcement by another organism*" [83].)

This is a radical departure from the generative view of human language, where language existed on its own and its understanding does not necessarily require the existence of the outside world nor the existence of a listener. It is no wonder why Chomsky was so harsh in criticizing Skinner's work in [27]. This departure, as I see it, is the departure toward a functional view of language. *Language is a function of communication*.

1.2.2 Language Understanding as a Function Approximation

Let's make a large jump here such that we consider this function as a mathematical function. This function (called language) takes as input the state of the surrounding world, the speaker's speech, either written, spoken or signed and the listener's mental

[2] Chomsky does not agree: "*it is wrong to think of human use of language as characteristically informative, in fact or in intention.*" [28].

state[3] Inside the function, the listener's mental state is updated to incorporate the new idea from the speaker's speech. The function then returns a response by the listener (which may include "no response" as well) and a set of non-verbal action sequences (what would be the action sequence if the speaker insulted the listener?).

In this case, language understanding, both from humans' and machines' perspective, boils down to figuring out the internal working of this function. In other words, we understand language by learning the internal mechanism of the function. Furthermore, this view suggests that the underlying structures of language are heavily dependent on the surrounding environment (context) as well as on the target task. The former (context dependence) is quite clear, as the function takes as input the context, but the latter may be confusing now. Hopefully, this will become clearer later in the course.

How can we approximate this function? How can we figure out the internal working mechanism of this function? What tools do we have?

Language Understanding by Machine Learning This functional view of languages suddenly makes machine learning a very appealing tool for understanding human languages. After all, function approximation is *the* core of machine learning. Classification is a classical example of function approximation, clustering is a function approximation where the target is not given, generative modeling learns a function that returns a probability of an input, and so on.

When we approximate a function in machine learning, the prime ingredient is data. We are given data which was either generated from this function (unsupervised learning) or well fit this function (supervised learning), based on which we adjust our approximation to the function, often iteratively, to best fit the data. But, I must note here that it does not matter how well the approximated function fits the data it was fitted to, but matters how well this approximation fits *unseen* data.[4]

In language understanding, this means that we collect a large data set of input and output pairs (or conversations together with the recording of the surrounding environment) and fit some arbitrary function to well predict the output given an input. We probably want to evaluate this approximation in a novel conversation. If this function makes a conversation just like a person, voilà, we made a machine that passed the Turing test. Simple, right?

Problem Unfortunately, as soon as we try to do this, we run into a big problem. This problem is not from machine learning nor languages, but the definition of this function of language.

Properly approximating this function requires us to either simulate or record the whole world (in fact, the whole universe.) For, this function takes as input and maintains as internal state the surrounding world (context) and the mental state of the individual (speaker.) This is unavoidable, if we wanted to very well approximate this function as a whole.

It is unclear, however, whether we want to approximate the full function. For a human to survive, yes, it is likely that the full function is needed. But, if our goal is

[3] We assume here that a such thing exists however it is represented in our brain.
[4] This is a matter of generalization, and we will talk about this more throughout the course.

restricted to a certain task (such as translation, language modelling, and so on), we may not want to approximate this function fully. We probably want to approximate only a subset of this whole function. For instance, if our goal is to understand the process of translation from one language to another, we can perhaps ignore all but the speech input to the function and all but the speech output from the function, because often a (trained) person can translate a sentence in one language to another without knowing the whole context.

This latter approach to language understanding–approximating a partial function of languages– will be at the core of this course. We will talk about various language tasks that are a part of this whole function of language. These tasks will include, but are not limited to, language modelling, machine translation, image/video description generation and question answering. For these tasks and potentially more, we will study how to use machine learning, or more specifically deep learning, to solve these tasks by approximating sub-functions of language.

Chapter 2

Function Approximation as Supervised Learning

Throughout this course, we will extensively use artificial neural networks[1] to approximate (a part of) the function of natural language. This makes it necessary for us to study the basics of neural networks first, and this lecture and a couple of subsequent ones are designed to serve this purpose.

2.1 Function Approximation: Parametric Approach

2.1.1 Expected Cost Function

Let us start by defining a data distribution p_{data}. p_{data} is defined over a pair of input and output vectors, $\mathbf{x} \in \mathbb{I}^d$ and $\mathbf{y} \in \mathbb{O}^k$, respectively. $\mathbb{I}$ and $\mathbb{O}$ are respectively sets of all possible input and output values, such as $\mathbb{R}$, $\{0,1\}$ and $\{0,1,\ldots,L\}$. This data distribution is not known to us.

The goal is to find a relationship between $\mathbf{x}$ and $\mathbf{y}$. More specifically, we are interested in finding a function $f : \mathbb{R}^d \to \mathbb{O}^k$ that generates the output $\mathbf{y}$ given its corresponding input $\mathbf{x}$. The very first thing we should do is to put some constraints on the function f to make our search for the correct f a bit less impossible. In this lecture, and throughout the course, I will consider only a parametric function f, in which case the function is fully specified with a set of parameters $\boldsymbol{\theta}$.

Next, we must define a way to measure how well the function f approximates the underlying mechanism of generation ($\mathbf{x} \to \mathbf{y}$). Let's denote by $\hat{\mathbf{y}}$ the output of the function with a particular set θ of parameters and a given input $\mathbf{x}$:

$$\hat{\mathbf{y}} = f_{\boldsymbol{\theta}}(\mathbf{x})$$

[1] From here on, I will simply drop artificial and call them neural networks. Whenever I say "neural network", it refers to artificial neural networks.

How well f approximates the true generating function is equivalent to how far $\hat{\mathbf{y}}$ is from the correct output $\mathbf{y}$. Let's use $D(\hat{\mathbf{y}}, \mathbf{y})$ for now call this distance[2] between $\hat{\mathbf{y}}$ and $\mathbf{y}$

It is clear that we want to find θ that minimizes $D(\hat{\mathbf{y}}, \mathbf{y})$ for every pair in the space $(RR^d \times \mathbb{O}^k)$. But, wait, every pair equally likely? Probably not, for we do not care how well f_θ approximates the true function, when a pair of input $\mathbf{x}$ and output $\mathbf{y}$ is unlikely, meaning we do not care how bad the approximation is, if $p_{\text{data}}(\mathbf{x}, \mathbf{y})$ is small. However, this is a bit difficult to take into account, as we must decided on the threshold below which we consider any pair irrelevant.

Hence, we *weight* the distance between the approximated $\hat{\mathbf{y}}$ and the correct $\mathbf{y}$ of each pair $(\mathbf{x}, \mathbf{y})$ in the space by its probability $p(\mathbf{x}, \mathbf{y})$. Mathematically saying, we want to find

$$\arg\min_\theta \int_{\mathbf{x}} \int_{\mathbf{y}} p_{\text{data}}(\mathbf{x}, \mathbf{y}) D(\hat{\mathbf{y}}, \mathbf{y}) d\mathbf{x} d\mathbf{y},$$

where the integral $\int$ should be replaced with the summation $\sum$ if any of $\mathbf{x}$ and $\mathbf{y}$ is discrete.

We call this quantity being minimized with respect to the parameters θ a cost function $C(\theta)$. This is equivalent to computing the *expected* distance between the predicted output $\hat{\mathbf{y}}$ and the correct one $\mathbf{y}$:

$$C(\theta) = \int_{\mathbf{x}} \int_{\mathbf{y}} p_{\text{data}}(\mathbf{x}, \mathbf{y}) D(\hat{\mathbf{y}}, \mathbf{y}) d\mathbf{x} d\mathbf{y}, \tag{2.1}$$

$$= \mathbb{E}_{(\mathbf{x}, \mathbf{y}) \sim p_{\text{data}}} \left[D(\hat{\mathbf{y}}, \mathbf{y}) \right] \tag{2.2}$$

This is often called an expected loss or risk, and minimizing this cost function is referred to as *expected risk minimization* [90].

Unfortunately $C(\theta)$ cannot be (exactly) computed for a number of reasons. The most important reason among them is simply that we don't know what the data distribution p_{data} is. Even if we have access to p_{data}, we can exactly compute $C(\theta)$ only with heavy assumptions on both the data distribution and the distance function.[3]

2.1.2 Empirical Cost Function

This does not mean that we are doomed from the beginning. Instead of the full-blown description of the data distribution p_{data}, we will assume that someone miraculously gave us a finite set of pairs drawn from the data distribution. We will call this a training set D:

$$D = \left\{ (\mathbf{x}^1, \mathbf{y}^1), \ldots, (\mathbf{x}^N, \mathbf{y}^N) \right\}.$$

As we have access to the samples from the data distribution, we can use Monte Carlo method to approximate the expected cost function $C(\theta)$ such that

$$C(\theta) \approx \tilde{C}(\theta) = \frac{1}{N} \sum_{n=1}^{N} D(\hat{\mathbf{y}^n}, \mathbf{y}^n). \tag{2.3}$$

[2] Note that we do not require this distance to satisfy the triangular inequality, meaning that it does not have to be a distance. However, I will just call it distance for now.

[3] Why?

We call this approximate $\tilde{C}(\theta)$ of the expected cost function, an empirical cost function (or empirical risk or empirical loss.)

Because empirical cost function is readily computable, we will mainly work with the empirical cost function not with the expected cost function. However, keep in mind that at the end of the day, the goal is to find a set of parameters that minimizes the *expected* cost.

2.2 Learning as Optimization

We often call this process of finding a good set of parameters that minimizes the expected cost *learning*. This term is used from the perspective of a machine which implements the function f_θ, as it *learns* to approximate the true generating function f from training data.

From what I have described so far, it may have become clear even without me mentioning that learning is *optimization*. We have a clearly defined function (the empirical cost function $\tilde{C}$) which needs to be minimized with respect to its input θ.

2.2.1 Gradient-based Local Iterative Optimization

There are many optimization algorithms one can use to find a set of parameters that minimizes $\tilde{C}$. Sometimes, you can even find the optimal set of parameters in a closed form equation.[4] In most cases, because there is no known closed-form solution, it is typical to use an iterative optimization algorithm (see [37] for in-depth discussion on optimization.)

By an *iterative* optimization, I mean an algorithm which refines its estimate of the optimal set of parameters little by little until the values of the parameters converge to the optimal (expected) cost function. Also, it is worthwhile to note that most iterative optimization algorithms are *local*, in the sense that they do not require us to evaluate the whole parameter space, but only a small subset along the path from the starting point to the convergence point.[5]

Here I will describe the simplest one among those local iterative optimization algorithms, called gradient descent (GD) algorithm. As the name suggests, this algorithm depends entirely on the gradient of the cost function.[6]

---

[4] One such example is a linear regression where

- $f_{\theta=\{\mathbf{W}\}}(\mathbf{x}) = \mathbf{W}\mathbf{x}$
- $D(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{2}\|\hat{\mathbf{y}} - \mathbf{y}\|^2$

In this case, the optimal $\mathbf{W}$ is

$$\mathbf{W} = \mathbf{Y}\mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top)^{-1}, \tag{2.4}$$

where

$$\mathbf{X} = \left[\mathbf{x}^1; \dots; \mathbf{x}^N\right], \mathbf{Y} = \left[\mathbf{y}^1; \dots; \mathbf{y}^N\right].$$

Try it yourself!

[5] There are *global* optimization algorithms, but they are out of scope for this course. See, for instance, [16] for one such algorithm called Bayesian optimization.

[6] From here on, I will use the cost function to refer to the *empirical* cost function.

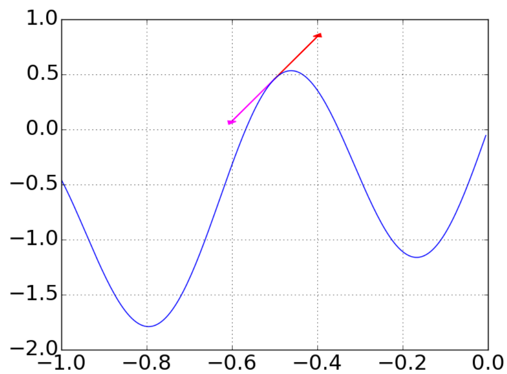


Figure 2.1: (blue) $f(x) = \sin(10x) + x$. (red) a gradient at $x = -0.6$. (magenta) a negative gradient at $x = -0.6$.

The gradient of a function $\nabla\tilde{C}$ is a vector whose direction points to the direction of the greatest rate of increase in the function's value and whose magnitude measures this rate. At each point θ_t in the parameter space, the gradient of the cost function $\nabla\tilde{C}(\theta_t)$ is the *opposite* direction toward which we want to move the parameters. See Fig. 2.1 for graphical illustration.

One important point of GD that needs to be mentioned here is on how large a step one takes each time. As clear from the magenta line (the direction opposite to the direction given by the gradient) in Fig. 2.1, if too large a step is taken toward the negative gradient direction, the optimization process will overshoot and miss the (local) minimum around $x = -0.8$. This step size, or sometimes called learning rate, η is one most important hyperparameter of the GD algorithm.

Now we have all the ingredients for the GD algorithm: $\nabla\tilde{C}$ and η. The GD algorithm iterates the following step:

$$\theta \leftarrow \theta - \eta\nabla\tilde{C}(\theta). \tag{2.5}$$

The iteration continues until a certain stopping criterion is met, which we will discuss shortly.

2.2.2 Stochastic Gradient Descent

This simple GD algorithm works surprisingly quite well, and it is a fundamental basis upon which many advanced optimization algorithms have been built. I will present a list of few of those advanced algorithms later on and discuss them briefly. But, before going into those advanced algorithms, let's solve one tiny, but significant issue of the GD algorithm.

This tiny, but significant issue arises especially often in machine learning. That is, it is computationally very expensive to compute $\tilde{C}$ and consequently its gradient $\nabla\tilde{C}$, thanks to the ever increasing size of the training set D.

Why is the growing size of the training set making it more and more computationally demanding to compute $\tilde{C}$ and $\nabla\tilde{C}$? This is because both of them are essentially the sum of as many per-sample costs as there are examples in the training set. In other words,

$$\tilde{C}(\theta) = \frac{1}{N}\sum_{n=1}^{N}\tilde{C}(\mathbf{x}^n, \mathbf{y}^n|\theta),$$

$$\nabla\tilde{C}(\theta) = \frac{1}{N}\sum_{n=1}^{N}\nabla\tilde{C}(\mathbf{x}^n, \mathbf{y}^n|\theta).$$

And, N goes up to millions or billions very easily these days.

This enormous computational cost involved in each GD step has motivated the *stochastic gradient descent* (SGD) algorithm [77, 13].

First, recall from Eq. (2.3) that the cost function we minimize is the *empirical* cost function $\tilde{C}$ which is the sample-based approximation to the *expected* cost function C. This approximation was done by assuming that the training examples were drawn randomly from the data distribution p_{data}:

$$C(\theta) \approx \tilde{C}(\theta) = \frac{1}{N}\sum_{n=1}^{N} D(\hat{\mathbf{y}^n}, \mathbf{y}^n).$$

In fact, as long as this assumption on the training set holds, we can always approximate the expected cost function with a fewer number of training examples:

$$C(\theta) \approx \tilde{C}_{\mathscr{M}}(\theta) = \frac{1}{|\mathscr{M}|}\sum_{m\in\mathscr{M}} D(\hat{\mathbf{y}^m}, \mathbf{y}^m),$$

where $M \ll N$ and $\mathscr{M}$ is the indices of the examples in this much smaller subset of the training set. We call this small subset a *minibatch*.

Similarly, this leads to a minibatch-based estimate of the gradient as well:

$$\nabla\tilde{C}_{\mathscr{M}}(\theta) = \frac{1}{|\mathscr{M}|}\sum_{m\in\mathscr{M}} \nabla D(\hat{\mathbf{y}^m}, \mathbf{y}^m).$$

It must now be clear to you where I am headed toward. At each GD step, instead of using the full training set, we will use a small subset $\mathscr{M}$ which is randomly selected to compute the gradient estimate. In other words, we use $\tilde{C}_{\mathscr{M}}$ instead of $\tilde{C}$, and $\nabla\tilde{C}_{\mathscr{M}}$ instead of $\nabla\tilde{C}$, in Eq. (2.5).

Because computing $\tilde{C}_{\mathscr{M}}$ and $\nabla\tilde{C}_{\mathscr{M}}$ is independent of the size of the training set, we can use SGD to make as many steps as we want without worrying about the growing size of training examples. This is highly beneficial, as regardless of how many training examples you used to compute the gradient, we can only take a tiny step toward that descending direction. Furthermore, the increased level of noisy in the gradient estimate due to the small sample size has been suspected to help reaching a better solution in high-dimensional non-convex problems (such as those in training deep neural networks) [61].[7]

[7] Why would this be the case? It is worth thinking about this issue further.

We can set M to be any constant, and in an extreme, we can set it to 1 as well. In this case, we call it online SGD.[8] Surprisingly, already in 1951, it was shown that using a single example each time is enough for the SGD to converge to a minimum (under certain conditions, obviously) [77].

This SGD algorithm will be at the core of this course and will be discussed further in the future lectures.

2.3 When do we stop learning?

From here on, I assume that we approximate the ground truth function by iteratively refining its set of parameters, in most cases using *stochastic gradient descent*. In other words, learning of a machine that approximates the true generating function f happens gradually as the machine goes over the training examples little by little over time.

Let us go over again what kind of constraints/issue we have first:

1. Lack of access to the expected cost function $C(\theta)$
2. Computationally expensive empirical cost function $\tilde{C}(\theta)$
3. (Potential) non-convexity of the empirical cost function $\tilde{C}(\theta)$

The most severe issue is that we do not have access to the expected cost function which is the one we want to minimize in order to work well with *any* pair of input $\mathbf{x}$ and output $\mathbf{y}$. Instead, we have access to the empirical cost function which is a finite sample approximation to the expected cost function.

Why is this a problem? Because, we do not have a guarantee that the (local) minimum of the empirical cost function corresponds to the (local) minimum of the expected cost function. An example of this mismatch between the expected and empirical cost functions is shown in Fig. 2.2.

As in the case shown in Fig. 2.2, it is not desirable to minimize the empirical cost function perfectly. The parameters that perfectly minimize the empirical cost function (in the case of Fig. 2.2, the slope a of a linear function $f(x) = ax$) will likely be a sub-optimal cost for the expected cost function about which we really care.

2.3.1 Early Stopping

What should we do? There are many ways to avoid this weird contradiction where we want to optimize the cost function well but not too well. Among those, one most important trick is *early stopping*, which is only applicable when iterative optimization is used.

First, we will split the training set D into two partitions D_{train} and D_{val}.[9] We call them a training set and a validation set, respectively. In practice it is a good idea to keep D much larger than D', because of the reasons that will become clear shortly.

[8] Okay, this is not true in a strict sense. SGD is an online algorithm with $M = 1$ originally, and using $M > 1$, is a variant of SGD, often called, minibatch SGD. However, as using minibatches ($M > 1$) is almost always the case in practice, I will refer to minibatch SGD as SGD, and to the original SGD as online SGD.

[9] Later on, we will split it further into three partitions.

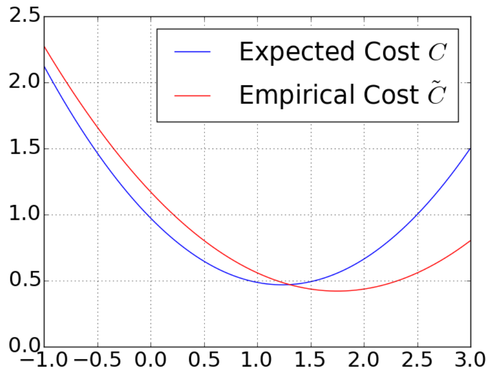


Figure 2.2: (blue) Expected cost function $C(\theta)$. (red) Empirical cost function $\tilde{C}(\theta)$. The underlying true generating function was $f(x) = \sin(10x) + x$. The cost function uses the squared Euclidean distance. The empirical cost function was computed based on 10 noisy examples of which x's were sampled from the uniform distribution between 0 and 1. For each sample input x, noise from zero-mean Gaussian distribution with standard deviation 0.01 was added to $f(x)$ to emulate the noisy measurement channel.

Further, let us define the training cost as

$$\tilde{C}(\theta) = C_{\text{train}}(\theta) = \frac{1}{|D_{\text{train}}|} \sum_{(x,y) \in D_{\text{train}}} D_{\text{train}}(\hat{y}, y), \tag{2.6}$$

and the validation cost as

$$C_{\text{val}}(\theta) = \frac{1}{|D_{\text{val}}|} \sum_{(x,y) \in D_{\text{val}}} D(\hat{y}, y). \tag{2.7}$$

With these two cost functions we are all ready to use early stopping now.

After every few updates using SGD (or GD), the validation cost function is evaluated with the current set of parameters. The parameters are updated (i.e., the training cost function is optimized) until the validation cost does not decrease, or starts to increase instead of decreasing.

That's it! It is almost free, as long as the size of the validation set is reasonable, since each evaluation is at most as expensive as computing the gradient of the empirical cost function. Because of the simplicity and effectiveness, this early stopping strategy has become *de facto* standard in deep learning and in general machine learning.

The question that needs to be asked here is what the validation cost function does here. Clearly, it approximates the expected cost function C, similarly to the empirical cost function $\tilde{C}$ as well as the training cost function C_{train}. In the infinite limit of the size of either training or validation set, they should coincide, but in the case of a finite set, those two cost functions differ by the noise in sampling (sampling pairs from the data distribution) and observation (noise in $\mathbf{y} = f(\mathbf{x})$.)

The fact that we explicitly optimize the training cost function implies that there is a possibility (in fact, almost surely in practice) that the set of parameters found by this optimization process may capture not only the underlying generating function but also noise in the observation and sampling procedure. This is an issue, because we want our machine to approximate the true generating function not the noise process involved.

The validation cost function measures both the true generating structure as well as noise injected during sampling and observation. However, assuming that noise is not correlated with the underlying generating function, noise introduced in the validation cost function differs from that in the training cost function. In other words, the set of parameters that perfectly minimizes the training cost function (thereby capturing even noise in the training set) will be penalized when measured by the validation cost function.

2.3.2 Model Selection

In fact, the use of the validation cost does not stop at the early stopping. Rather, it has a more general role in model selection. First, we must talk about model selection itself.

This whole procedure of optimization, or learning, can be cast as a process of searching for the best *hypothesis* over the entire space $\mathscr{H}$ of hypotheses. Here, each hypothesis corresponds to each possible function (with a unique set of parameters and a unique functional form) that takes the input $\mathbf{x}$ and output $\mathbf{y}$. In the case of regression ($\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{y} \in \mathbb{R}$), the hypothesis space includes an n-th order polynomial function

$$f(x) = \sum_{\sum_{k=1}^{d} i_k = n, i_k \geq 0} a_{i_1, i_2, \ldots, i_k} \prod_{k'=1}^{d} x_{k'}^{i_k},$$

where $a_{i_1, i_2, \ldots, i_k}$'s are the coefficients, and any other functional form that you can imagine as long as it can process $\mathbf{x}$ and return a real-valued scalar. In the case of neural networks, this space includes all the possible model architectures which are defined by the number of layers, the type of nonlinearities, the number of hidden units in each layer and so on.

Let us use $M \in \mathscr{H}$ to denote one hypothesis.[10] One important thing to remember is that the parameter space is only a subset of the hypothesis space, because the parameter space is defined by a family of hypotheses (the parameter space of a linear function cannot include a set of parameters for a second-order polynomial function.)

Given a definition of expected cost function, we can *score* each hypothesis M by the corresponding cost C_M. Then, the whole goal of function approximation boils down to the search for a hypothesis M with the minimal expected cost function C. But, of course, we do not have access to the expected cost function and resort to the empirical cost function based on a given training set.

The optimization-based approach we discussed so far searches for the best hypothesis based on the empirical cost iteratively. However, because of the issue of *overfitting* which means that the optimization algorithm overshot and missed the local minimum of the expected cost function (because it was aimed at the local minimum of the empirical cost function), I introduced the concept of early stopping based on the validation cost.

[10] M, because each hypothesis corresponds to one learning *machine*.

This is unfortunately not satisfactory, as we have only searched for the best hypothesis inside a small subset of the whole hypothesis space $\mathscr{H}$. What if another subset of the hypothesis space includes a function that better suits the underlying generating function f? Are we doomed?

It is clearly better to try more than one subsets of the hypothesis space. For instance, for a regression task, we can try linear functions ($\mathscr{H}_1$), quadratic (second-order polynomial) functions ($\mathscr{H}_2$) and sinusoidal functions ($\mathscr{H}_3$). Let's say for each of these subsets, we found the best hypothesis (using iterative optimization and early stopping); $M_{\mathscr{H}_1}$, $M_{\mathscr{H}_2}$ and $M_{\mathscr{H}_3}$. Then, the question is how we should choose one of those hypotheses.

Similar to what we've done with early stopping, we can use the validation cost to compare these hypotheses. Among those three we choose one that has the smallest validation cost $C_{\text{val}}(M)$.

This is one way to do *model selection*, and we will talk about another way to do this later.

2.4 Evaluation

But, wait, if this is an argument for using the validation cost to *early stop* the optimization (or learning), one needs to notice something weird. What is it?

Because we used the validation cost to stop the optimization, there is a chance that the set of parameters we found is optimal for the validation set (whose structure consists of both the true generating function and sampling/observation noise), but not to the general data distribution. This means that we cannot tell whether the function estimate $\hat{f}$ approximating the true generating function f is a good fit by simply early stopping based on the validation cost. Once the optimization is done, we need yet another metric to see how well the learned function estimate $\hat{f}$ approximates f.

Therefore, we need to split the training set not into two partitions but into *three* partitions. We call them a training set D_{train}, a validation set D_{val} and a test set D_{test}. Consequently, we will have three cost functions; a training cost function C_{train}, a validation cost function C_{val} and a test cost function C_{test}, similarly to Eqs. 2.6–2.7.

This test cost function is the one we use to compare different hypotheses, or models, fairly. Any hypothesis that worked best in terms of the test cost is the one that you choose.

Let's not Cheat One most important lesson here is that you *must never look at a test set*. As soon as you take a peak at the test set, it will influence your choice in the model structure as well as any other hyperparameters biasing toward a better test cost. The best option is to never ever look at the test set until it is absolutely needed (e.g., need to present your result.)

2.5 Linear Regression for Non-Linear Functions

Let us start with a simple linear function to approximate a true generating function such that

$$\hat{\mathbf{y}} = f(\mathbf{x}) = \mathbf{W}^\top \mathbf{x},$$

where $\mathbf{W} \in \mathbb{R}^{d \times l}$ is the weight matrix. In this case, this weight matrix is the only parameter, i.e., $\theta = \{\mathbf{W}\}$.

The empirical cost function is then

$$\tilde{C}(\theta) = \frac{1}{N} \sum_{n=1}^{N} \frac{1}{2} \left\| \mathbf{y}^n - \mathbf{W}^\top \mathbf{x}^n \right\|_2^2.$$

The gradient of the empirical cost function is

$$\nabla \tilde{C}(\theta) = -\frac{1}{N} \sum_{n=1}^{N} \left(\mathbf{y}^n - \mathbf{W}^\top \mathbf{x}^n \right)^\top \mathbf{x}^n. \tag{2.8}$$

With these two well defined, we can use the iterative optimization algorithm, such as GD or SGD, to find the best $\mathbf{W}$ that minimizes the empirical cost function.[11] Or, better is to use a validation set to stop the optimization algorithm at the point of the minimal validation cost function (remember early stopping?)

Now, but we are not too satisfied with a linear network, are we?

2.5.1 Feature Extraction

Why are we not satisfied?

First, we are not sure whether the true generating function f was a linear function. If it is not, can we expect linear regression to approximate the true function well? Of course, not. We will talk about this shortly.

Second, because we were *given* $\mathbf{x}$ (meaning we did not have much control over what we want to measure as $\mathbf{x}$), it is unclear how well $\mathbf{x}$ represents the input. For instance, consider doing a sales forecast of air conditioner at one store which opened five years ago. The input x is the number of days since the opening date of the store (1 Jan 2009), and the output y is the number of units sold on each day.

Clearly, in this example, the relationship between x and y is not linear. Furthermore, perhaps the most important feature for predicting the sales of air conditioners is missing from the input x, which is a month (or a season, if you prefer.) It is likely that the sales bottoms out during the winter (perhaps sometime around December, January and February,) and it hits the peak during summer months (around May, June and July.) In other words, if we look at how far the month is away from July, we can predict the sales quite well even with linear regression.

[11] In fact, looking at Eq. (2.8), it's quite clear that you can compute the optimal $\mathbf{W}$ analytically. See Eq. (2.4).

Let us call this quantity $\phi(x)$, or equivalent *feature*, such that

$$\phi(x) = |m(x) - \alpha|, \tag{2.9}$$

where $m(x) \in \{1, 2, \ldots, 12\}$ is the month of x and $\alpha = 5.5$. With this feature, we can fit linear regression to better approximate the sales figure of air conditioners. Furthermore, we can add yet another feature to improve the predictive performance. For instance, one such feature can be which day of week x is.

This whole process of extracting a good set of features that will make our choice of parametric function family (such as linear regression in this case) is called *feature extraction*. This feature extraction is an important step in machine learning and has often been at the core of many applications such as computer vision (the representative example is SIFT [64].)

Feature extraction often requires heavy knowledge of the domain in which this function approximation is applied. To use linear regression for computer vision, it is a good idea to use computer vision knowledge to extract a good set of features. If we want to use it for environmental problems, we must first notice which features must be important and how they should be represented for linear regression to work.

This is okay for a machine learning practitioner in a particular field, because the person has in-depth knowledge about the field. There are however many cases where there's simply not enough domain knowledge to exploit. To make the matter worse, it is likely that the domain knowledge is not correct, making the whole business of using manually extracted features futile.

Chapter 3

Neural Networks and Backpropagation Algorithm

3.1 Conditional Distribution Approximation

I have mainly described so far as if the function we approximate or the function we use to approximate returns only a constant value, as in one point $\mathbf{y}$ in the output space. This is however not true, and in fact, the function can return anything including a *distribution* [15, 32, 11].

Let's first decompose the data distribution p_{data} into the product of two terms:

$$p_{\text{data}}(\mathbf{x}, \mathbf{y}) = p_{\text{data}}(\mathbf{x}) p_{\text{data}}(\mathbf{y}|\mathbf{x}).$$

It becomes clear that one way to sample from p_{data} is to sample an input $\mathbf{x}^n$ from $p_{\text{data}}(\mathbf{x})$ and subsequently sample the corresponding output $\mathbf{y}^n$ from the conditional distribution $p_{\text{data}}(\mathbf{y}|\mathbf{x}^n)$.

This implies that the function approximation of the generating function $(f : \mathbf{x} \to \mathbf{y})$ is effectively equivalent to approximating the conditional distribution $p_{\text{data}}(\mathbf{y}|\mathbf{x})$. This may suddenly sound much more complicated, but it should not alarm you at all. As long as we choose to use a distribution parametrized by a small number of parameters to approximate the conditional distribution $p_{\text{data}}(\mathbf{y}|\mathbf{x})$, this is quite manageable without almost any modification to the expected and empirical cost functions we have discussed.

Let us use $\theta(\mathbf{x})$ to denote a set of parameters for the probability distribution $\tilde{p}(\mathbf{y}|\mathbf{x}, \theta(\mathbf{x}))$ approximating the true, underlying probability distribution $p_{\text{data}}(\mathbf{y}|\mathbf{x})$. As the notation suggests, the function now returns *the parameters of the distribution* $\theta(\mathbf{x})$ given the input $\mathbf{x}$.

For example, let's say $\mathbf{y} \in \{0, 1\}^k$ is a binary vector and we chose to use independent Bernoulli distribution to approximate the conditional distribution $p_{\text{data}}(\mathbf{y}|\mathbf{x})$. In this case, the parameters that define the conditional distribution are the means of k

dimensions:

$$\tilde{p}(\mathbf{y}|\mathbf{x}) = \prod_{k'=1}^{k} p(y_{k'}|\mathbf{x}) = \prod_{k'=1}^{k} \mu_{k'}^{y_{k'}} (1-\mu_{k'})^{1-y_{k'}}. \tag{3.1}$$

Then the function $\theta(\mathbf{x})$ should output a k-dimensional vector of which each element is between 0 and 1.

Another example: let's say $\mathbf{y} \in \mathbb{R}^k$ is a real-valued vector. It is quite natural to use a Gaussian distribution with a diagonal covariance matrix to approximate the conditional distribution $p(\mathbf{y}|\mathbf{x})$:

$$\tilde{p}(\mathbf{y}|\mathbf{x}) = \prod_{k'=1}^{k} \frac{1}{\sqrt{2\pi}\sigma_{k'}} \exp\left(\frac{(y_{k'}-\mu_{k'})^2}{2\sigma_{k'}^2}\right). \tag{3.2}$$

The parameters for this conditional distribution are $\theta(\mathbf{x}) = \{\mu_1, \mu_2, \ldots, \mu_k, \sigma_1, \sigma_2, \ldots, \sigma_k\}$, where $\mu_k \in \mathbb{R}$ and $\sigma_k \in \mathbb{R}_{>0}$.

In this case of probability approximation, it is natural to use Kullback-Leibler (KL) divergence to measure the distance.[1] The KL divergence from one distribution P to the other Q is defined[2] by

$$\mathrm{KL}(P\|Q) = \int P(\mathbf{x}) \log \frac{P(\mathbf{x})}{Q(\mathbf{x})} \mathrm{d}\mathbf{x}.$$

In our case of function/distribution approximation, we want to minimize the KL divergence from the data distribution $p_{\text{data}}(\mathbf{y}|\mathbf{x})$ to the approximate distribution $\tilde{p}(\mathbf{y}|\mathbf{x})$ averaged over the data distribution $p_{\text{data}}(\mathbf{x})$:

$$C(\theta) = \int p_{\text{data}}(\mathbf{x}) \mathrm{KL}(p_{\text{data}}\|\tilde{p}) \mathrm{d}\mathbf{x} = \int p_{\text{data}}(\mathbf{x}) \int p_{\text{data}}(\mathbf{y}|\mathbf{x}) \log \frac{p_{\text{data}}(\mathbf{y}|\mathbf{x})}{\tilde{p}(\mathbf{y}|\mathbf{x})} \mathrm{d}\mathbf{y}\mathrm{d}\mathbf{x}.$$

But again we do not have access to p_{data} and cannot compute this expected cost function.

Similarly to how we defined the empirical cost function earlier, we must approximate this expected KL divergence using the training set:

$$\tilde{C}(\theta) = \frac{1}{N} \sum_{n=1}^{N} -\log \tilde{p}(\mathbf{y}^n|\mathbf{x}^n). \tag{3.3}$$

As an example, if we choose to return the binary vector $\mathbf{y}$ as in Eq. (3.1), the empirical cost function will be

$$\tilde{C}(\theta) = -\frac{1}{N} \sum_{n=1}^{N} \sum_{k'=1}^{k} y_{k'} \log \mu_{k'} + (1-y_{k'}) \log(1-\mu_{k'}),$$

[1] Again, we use a loose definition of the distance where triangular inequality is not enforced.

[2] Why don't I say the KL divergence between two distributions here? Because, the KL divergence is not a symmetric measure, i.e., $\mathrm{KL}(P\|Q) \neq \mathrm{KL}(Q\|P)$.

which is often called a *cross entropy cost*. In the case of Eq. (3.2),

$$\tilde{C}(\theta) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k'=1}^{k}\frac{(y_{k'}-\mu_{k'})^2}{2\sigma_{k'}^2} - \log\sigma_{k'}. \tag{3.4}$$

Do you see something interesting in Eq. (3.4)? If we assume that the function outputs 1 for all $\sigma_{k'}$'s, we see that this cost function reduces to that using the Euclidean distance between the true output $\mathbf{y}$ and the mean μ. What does this mean?

There will be many occasions later on to discuss more about this perspective when we discuss language modelling. However, one thing we must keep in our mind is that there is nothing different between approximating a function and a distribution.

3.1.1 Why do we want to do this?

Before we move on to the main topic of today's lecture, let's try to understand why we want to output the distribution. Unlike returning a single point in the space, the distribution returned by the function f incorporates both the most likely outcome $\hat{y}$ as well as the uncertainty associated with this value.

In the case of the Gaussian output in Eq. (3.2), the standard deviation $\sigma_{k'}$, or the variance $\sigma_{k'}^2$, indicates how uncertain the function is about the output centered at $\mu_{k'}$. Similarly, the mean $\mu_{k'}$ of the Bernoulli output in Eq. (3.1) is directly proportional to the function's confidence in predicting that the k'-th dimension of the output is 1.

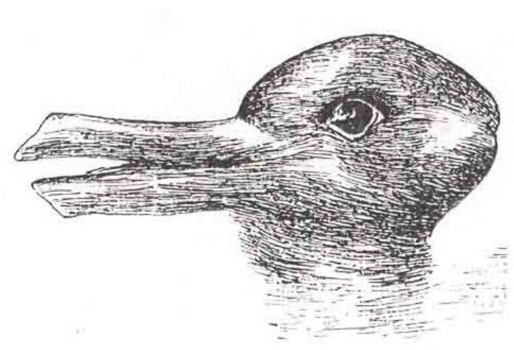


Figure 3.1: Is this a duck or a rabbit? [58] At the end of the day, we want our function f to return a conditional distribution saying that $p(\text{duck}|\mathbf{x}) = p(\text{rabbit}|\mathbf{x})$, instead of returning *the* answer out of these two possible answers.

This is useful in many aspects, but one important aspect is that it reflects the natural uncertainty of the underlying generating function. One input $\mathbf{x}$ may be interpreted in more than one ways, leading to two possible outputs, which happens more often than not in the real world. For instance, the famous picture in Fig. 3.1 can be viewed as a picture of a duck or a picture of a rabbit, in which case the function needs to output the probability distribution by which the same probability mass is assigned to both a duck and a rabbit. Furthermore, there is observational noise that cannot easily be identified and ignored by the function, in which case the function should return the uncertainty due to the observational noise along with the most likely (or the average) prediction.

3.1.2 Other Distributions

I have described two distributions (densities) that are widely used:

- Bernoulli distribution: binary classification
- Gaussian distribution: real value regression

Here, let me present one more distribution which we will use almost everyday through this course.

Categorical Distribution: Multi-Class Classification Multi-class classification is a task in which each example belongs to one of K classes. For each input x, the problem reduces to find a probability $p_k(x)$ of the k-th class under the constraint that

$$\sum_{k=1}^{K} p_k(x) = 1$$

It is clear that in this case, the function f returns K values $\{\mu_1, \mu_2, \ldots, \mu_K\}$, each of which is between 0 and 1. Furthermore, the sum of μ_k's must sum to 1. This can be achieved easily by letting f to compute affine transformation of x (or $\phi(x)$) to return K (unbounded) real values followed by a so called *softmax* function [15]:

$$\mu_k = \frac{\exp(w_k^\top \phi(x) + b_k)}{\sum_{k'=1}^{K} \exp(w_{k'}^\top \phi(x) + b_k)}, \tag{3.5}$$

where $w_k \in \mathbb{R}^{\dim(\phi(x))}$ and $b_k \in \mathbb{R}$ are the parameters of affine transformation.

In this case, the (empirical) cost function based on the KL divergence is

$$C(\theta) = -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \mathbb{I}_{k=y^n} \mu_k, \tag{3.6}$$

where

$$\mathbb{I}_{k=y^n} = \begin{cases} 1, & \text{if } k = y^n \\ 0, & \text{otherwise} \end{cases} \tag{3.7}$$

3.2 Feature Extraction is also a Function

We talked about the manual feature extraction in the previous lecture (see Sec. 2.5.1. But, this is quite unsatisfactory, because this whole process of manual feature extraction is heavily dependent on the domain knowledge, meaning that we cannot have a generic principle on which we design features. This raises a question: instead of manually designing features ourselves, is it possible for this to happen automatically?

One thing we notice is that the feature extraction process $\phi(\mathbf{x})$ is nothing but a *function*. A function of a function is a function, right? In other words, we will extend our definition of the function to include the feature extraction function:

$$\hat{\mathbf{y}} = f(\phi(\mathbf{x})).$$

We will assume that the feature extraction function ϕ is also parametrized, and its parameters are included in the set of parameters which includes those of f. As an example, α in Eq. (2.9) is a parameter of the feature extraction ϕ.

A natural next question is which family of parametric functions we should use for ϕ. We run into the same issue we talked about earlier in Sec. 2.3: the size of hypothesis space is simply too large!

Instead of choosing one great feature extraction function, we can go for a stack of simple transformations which are all learned.[3] Each transformation can be as simple as affine transformation followed by a simple point-wise nonlinearity:

$$\phi_0(\mathbf{x}) = g(\mathbf{W}_0\mathbf{x} + \mathbf{b}_0), \tag{3.8}$$

where $\mathbf{W}_0$ is the weight matrix, $\mathbf{b}_0$ is the bias and g is a point-wise nonlinearity such as tanh.[4]

One interesting thing is that if the dimensionality of the transformed feature vector $\phi_0(\mathbf{x})$ is *much* larger than that of $\mathbf{x}$, the function $f(\phi_0(\mathbf{x}))$ can approximate any function from $\mathbf{x}$ to $\mathbf{y}$ under some assumptions, even when the parameters $\mathbf{W}_0$ and $\mathbf{b}_0$ are randomly selected! [31]

The problem solved, right? We just put a huge matrix $\mathbf{W}_0$, apply some nonlinear function g to it and fit linear regression as I described earlier. We don't even need to touch $\mathbf{W}_0$ and $\mathbf{b}_0$. All we need to do is replace the input $\mathbf{x}^n$ of all the pairs in the training set to $\phi_0(\mathbf{x}^n)$.

In fact, there is a group of researchers claiming to have figured this out by themselves less than a decade ago (as of 2015) who call this model an *extreme learning machine* [48]. There have been some debates about this so-called extreme learning machine. Here I will not make any comment myself, but would be a good exercise for you to figure out why there has been debates about this.

But, regardlessly, this is not what we want.[5] What we want is to fully tune the whole thing.

3.3 Multilayer Perceptron

The basic idea of multilayer perceptron is to stack a large number of those feature extraction *layers* in Eq. (3.8) between the input and the output. This idea is as old as the whole field of neural network research, dating back to early 1960s [78]. However, it took many more years for people to figure out a way to tune the whole network, both f and ϕ's together. See [80] and [60], if you are interested in the history.

[3] A great article about this was posted recently in `http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/`.

[4] Some of the widely used nonlinearities are

- Sigmoid: $\sigma(x) = \frac{1}{1+\exp(-x)}$
- Hyperbolic function: $\tanh(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}$
- Rectified linear unit: $\text{rect}(x) = \max(0, x)$

[5] And, more importantly, I will not accept any final project proposal whose main model is based on the ELM.

3.3.1 Example: Binary classification with a single hidden unit

Let us start with the simplest example. The input $x \in \mathbb{R}$ is a real-valued scalar, and the output $y \in \{0, 1\}$ is a binary value corresponding to the input's label. The feature extractor ϕ is defined as

$$\phi(x) = \sigma(ux + c), \tag{3.9}$$

where u and c are the parameters. The function f returns the mean of the Bernoulli conditional distribution $p(y|x)$:

$$\mu = f(x) = \sigma(w\phi(x) + b). \tag{3.10}$$

In both of these equations, σ is a sigmoid function:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \tag{3.11}$$

We use the KL divergence to measure the distance between the true conditional distribution $p(y|x)$ and the predicted conditional distribution $\hat{p}(y|x)$.

$$\begin{aligned}
\mathrm{KL}(p \| \hat{p}) &= \sum_{y \in \{0,1\}} p(y|x) \log \frac{p(y|x)}{\hat{p}(y|x)} \\
&= \sum_{y \in \{0,1\}} p(y|x) \log p(y|x) - p(y|x) \log \hat{p}(y|x).
\end{aligned}$$

Note that the first term in the summation $p(y|x) \log p(y|x)$ can be safely ignored in our case. Why? Because, this does not concern $\tilde{p}$ which is one we change in order to minimize this KL divergence.

Let's approximate this KL divergence with a single sample from $p(y|x)$ and leave only the relevant part. We will call this a per-sample cost:

$$\begin{aligned}
C_x &= -\log \hat{p}(y|x) && (3.12) \\
&= -\log \mu^y (1 - \mu)^{1-y} && (3.13) \\
&= -y \log \mu - (1 - y) \log(1 - \mu), && (3.14)
\end{aligned}$$

where μ is from Eq. (3.10). It is okay to work with this per-sample cost function instead of the full cost function, because the full cost function is almost always the (unweighted) sum of these per-sample cost functions. See Eq. (2.3).

We now need to compute the gradient of this cost function C_x with respect to all the parameters w, b, u and c. First, let's start with w:

$$\frac{\partial C_x}{\partial w} = \frac{\partial C_x}{\partial \mu} \frac{\partial \mu}{\partial \underline{\mu}} \frac{\partial \underline{\mu}}{\partial w},$$

which is a simple application of chain rule of derivatives. Compare this to

$$\frac{\partial C_x}{\partial b} = \frac{\partial C_x}{\partial \mu} \frac{\partial \mu}{\partial \underline{\mu}} \frac{\partial \underline{\mu}}{\partial b}.$$

In both equations, $\underline{\mu} = w\phi(x) + b$ which is the input to f.

Both of these derivatives *share* $\frac{\partial C_x}{\partial \mu} \frac{\partial \mu}{\partial \underline{\mu}}$, where

$$\frac{\partial C_x}{\partial \mu} \underbrace{\frac{\partial \mu}{\partial \underline{\mu}}}_{=\mu'} = -\frac{y}{\mu}\mu' + \frac{1-y}{1-\mu}\mu' = \frac{-y + y\mu + \mu - y\mu}{\mu(1-\mu)}\mu' = \frac{\mu - y}{\mu(1-\mu)}\mu' = \mu - y, \tag{3.15}$$

because the derivative of the sigmoid function $\frac{\partial \mu}{\partial \underline{\mu}}$ is

$$\mu' = \mu(1-\mu).$$

Note that this corresponds to computing the difference between the correct label y and the predicted label (probability) μ.

Given this output derivative $\frac{\partial C_x}{\partial \underline{\mu}}$, all we need to compute are

$$\frac{\partial \underline{\mu}}{\partial w} = \phi(x)$$
$$\frac{\partial \underline{\mu}}{\partial b} = 1.$$

From these computations, we see that

$$\frac{\partial C_x}{\partial w} = (\mu - y)\phi(x), \tag{3.16}$$
$$\frac{\partial C_x}{\partial b} = (\mu - y). \tag{3.17}$$

Let us continue on to u and c. We can again rewrite the derivatives w.r.t. these into

$$\frac{\partial C_x}{\partial u} = \frac{\partial C_x}{\partial \underline{\mu}} \frac{\partial \underline{\mu}}{\partial \phi} \frac{\partial \phi}{\partial \underline{\phi}} \frac{\partial \underline{\phi}}{\partial u}$$
$$\frac{\partial C_x}{\partial c} = \frac{\partial C_x}{\partial \underline{\mu}} \frac{\partial \underline{\mu}}{\partial \phi} \frac{\partial \phi}{\partial \underline{\phi}} \frac{\partial \underline{\phi}}{\partial c},$$

where $\underline{\phi}$ is the input to ϕ similarly to $\underline{\mu}$ was to the input to μ.

There are two things to notice here. First, we already have $\frac{\partial C_x}{\partial \underline{\mu}}$ from computing the derivatives w.r.t. w and b, meaning there is no need to re-compute it. Second, $\frac{\partial \mu}{\partial \underline{\phi}}$ is shared between the derivatives w.r.t. u and c.

Therefore, we first compute $\frac{\partial \underline{\mu}}{\partial \underline{\phi}}$:

$$\frac{\partial \underline{\mu}}{\partial \phi} \underbrace{\frac{\partial \phi}{\partial \underline{\phi}}}_{=\phi'} = w\phi' = w\phi(x)(1 - \phi(x))$$

Next, we compute

$$\frac{\partial \phi}{\partial u} = x$$
$$\frac{\partial \phi}{\partial c} = 1.$$

Now all the ingredients are there:

$$\frac{\partial C_x}{\partial u} =(\mu - y)w\phi(x)(1-\phi(x))x$$
$$\frac{\partial C_x}{\partial c} =(\mu - y)w\phi(x)(1-\phi(x)).$$

The most important lession to learn from here is that most of the computations needed to get the derivatives in this seemingly complicated multilayered computational graph (multilayer perceptron) are *shared*. At the end of the day, the amount of computation needed to compute the gradient of the cost function w.r.t. all the parameters in the network is only as expensive as computing the cost function itself.

3.3.2 Example: Binary classification with more than one hidden units

Let us try to generalize this simple, or rather simplest model, into a slightly more general setting. We will still look at the binary classification but with multiple hidden units and a multidimensional input such that:

$$\phi(x) = Ux + c,$$

where $U \in \mathbb{R}^{l\times d}$ and $c \in \mathbb{R}^l$. Consequently, w will be a l-dimensional vector.

The output derivative $\frac{\partial C_x}{\partial \mu}\frac{\partial \mu}{\partial \underline{\mu}}$ stays same as before. See Eq. (3.15). However, we note that the derivative of $\underline{\mu}$ with respect to w should now differ, because it's a vector.[6] Let's look at what this means.

The $\underline{\mu}$ can be expressed as

$$\underline{\mu} = w^\top \phi(x) + b = \sum_{i=1}^{l} w_i\phi_i(x) + b. \tag{3.18}$$

In this case, we can start computing the derivative with respect to each element of w_i separately:

$$\frac{\partial \underline{\mu}}{\partial w_i} = \phi_i(x),$$

[6] The Matrix Cookbook [74] is a good reference for this section.

and will put them into a vector:

$$\frac{\partial \underline{\mu}}{\partial w} = \left[\frac{\partial \underline{\mu}}{\partial w_1}, \frac{\partial \underline{\mu}}{\partial w_2}, \ldots, \frac{\partial \underline{\mu}}{\partial w_l}\right]^\top = [\phi_1(x), \phi_2(x), \ldots, \phi_l(x)]^\top = \phi(x)$$

Then, the derivative of the cost function C_y with respect to w can be written as

$$\frac{\partial C_y}{\partial w} = (\mu - y)\phi(x),$$

in which case nothing really changed from the case of a single hidden unit in Eq. (3.16).

Now, let's look at $\frac{\partial C_y}{\partial \phi}$. Again, because $\phi(x)$ is now a vector, there has to be some changes. Because $\frac{\partial C_y}{\partial \underline{\mu}}$ is already computed, we only need to look at $\frac{\partial \underline{\mu}}{\partial \phi}$. In fact, the procedure for computing this is identical to that for computing $\frac{\partial \underline{\mu}}{\partial w}$ due to the symmetry in Eq. (3.18). That is,

$$\frac{\partial \underline{\mu}}{\partial \phi} = w$$

Next, what about $\frac{\partial \phi}{\partial \underline{\phi}}$? Because the nonlinear activation function σ is applied element-wise, we can simply compute this derivative for each element in $\phi(x)$ such that

$$\frac{\partial \phi}{\partial \underline{\phi}} = \mathrm{diag}\left(\left[\phi_1'(x), \phi_2'(x), \ldots, \phi_l'(x)\right]^\top\right),$$

where diag returns a diagonal matrix of the input vector. In short, we will denote this as ϕ'

Overall so far, we have got

$$\frac{\partial C_y}{\partial \underline{\phi}} = (\mu - y)w^\top \phi'(x) = (\mu - y)(w \odot \mathrm{diag}(\phi'(x))),$$

where $\odot$ is an element-wise multiplication.

Now it is time to compute $\frac{\partial \underline{\phi}}{\partial U}$:

$$\frac{\partial \underline{\phi}}{\partial U} = \frac{\partial U^\top x}{\partial U} = x,$$

according to the Matrix Cookbook [74]. Then, let's look at the whole derivative w.r.t. U:

$$\frac{\partial C_y}{\partial U} = (\mu - y)(w \odot \mathrm{diag}(\phi'(x)))x^\top.$$

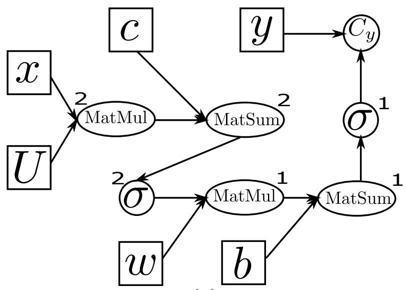
Note that all the vectors in this lecture note are *column* vectors.

For c, it's straightforward, since

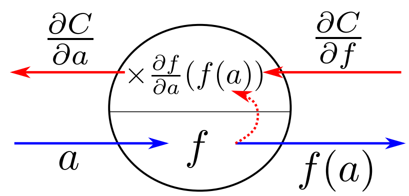
$$\frac{\partial \underline{\phi}}{\partial c} = 1.$$

3.4 Automating Backpropagation

This procedure, presented as two examples, is called a *backpropagation* algorithm. If you read textbooks on neural networks, you see a fancier way to explain this backpropagation algorithm by introducing a lot of fancy terms such as *local error* δ and so on. But, personally I find it much easier to understand backpropagation as a clever application of the chain rule of derivatives to a directed acyclic graph (DAG) in which each node computes a certain function ϕ using the output of the previous nodes. I will refer to this DAG as a computational graph from here on.



(a)



(b)

Figure 3.2: (a) A graphical representation of the computational graph of the example network from Sec. 3.3.2. (b) A graphical illustration of a function node ($\rightarrow$: forward pass, $\leftarrow$: backward pass.)

A typical computational graph looks like the one in Fig. 3.2 (a). This computational graph has two types of nodes; (1) function node (◯) and (2) variable node (□). There are four different types of function nodes; (1) $\text{MatMul}(A,B) = AB$, (2) $\text{MatSum}(A,B) = A+B$, (3) σ: element-wise sigmoid function and (4) C_y: cost node. The variables nodes correspond to either parameters or data (x and y.) Each function node has a number associated with it to distinguish between the nodes of the same function.

Now, in this computational graph, let us start computing the gradient using the backpropagation algorithm. We start from the last code, C_y, by computing $\frac{\partial C_y}{\partial y}$ and $\frac{\partial C_y}{\partial \sigma^1}$. Then, the function node σ^1 will compute its own derivative $\frac{\partial \sigma^1}{\partial \text{MatSum}^1}$ and multiply it with $\frac{\partial C_y}{\partial \sigma^1}$ passed *back* from the function node C_y. So far we've computed

$$\frac{\partial C_y}{\partial \text{MatSum}^1} = \frac{\partial C_y}{\partial \sigma^1} \frac{\partial \sigma^1}{\partial \text{MatSum}^1} \tag{3.19}$$

The function node MatSum^1 has two inputs b and the output of MatMul^1. Thus, this node computes two derivatives $\frac{\partial \text{MatSum}^1}{\partial b}$ and $\frac{\partial \text{MatSum}^1}{\partial \text{MatMul}^1}$. Each of these is multiplied with the backpropagated derivative $\frac{\partial C_y}{\partial \text{MatSum}^1}$ from Eq. (3.19). At this point, we already have the derivative of the cost function C_y w.r.t. one of the parameters b:

$$\frac{\partial C_y}{\partial b} = \frac{\partial C_y}{\partial \text{MatSum}^1} \frac{\partial \text{MatSum}^1}{\partial b}$$

This process continues mechanically until the very beginning of the graph (a set of root variable nodes) is reached. All we need in this process of backpropagating the derivatives is that each function node implements both *forward computation* as well as *backward computation*. In the backward computation, the function node received the derivative from the next function node, evaluates its own derivative with respect to the inputs (at the point of the forward activation) and passes theses derivatives to the corresponding previous nodes. See Fig. 3.2 (b) for the graphical illustration.

Importantly, the inner mechanism of a function node does not change depending on its context (or equivalently where the node is placed in a computational graph.) In other words, if each type of function nodes is implemented in advance, it becomes trivial to build a complicated neural network (including multilayer perceptrons) and compute the gradient of the cost function (which is one such function node in the graph) with respect to all the parameters as well as all the inputs.

This is a special case, called the reverse mode, of automatic differentiation.[7] It is probably the most valuable tool in deep learning, and fortunately many widely used toolkits such as Theano [9, 3] have implemented this reverse mode of automatic differentiation with an extensive number of function nodes used in deep learning everyday.

Before finishing this discussion on automating backpropagation, I'd like you to think of pushing this even further. For instance, you can think of each function node returning not its numerical derivative on its backward pass, but a computational subgraph computing its derivative. This means that it will return a *computational graph of gradient*, where the output is the derivatives of all the variable nodes (or a subset of them.) Then, we can use the same facility to compute the second-order derivatives, right?

3.4.1 What if a Function is *not* Differentiable?

From the description so far, one thing we notice is that backpropagation works only when each and every function node (in a computational graph) is differentiable. In other words, the nonlinear activation function must be chosen such that almost everywhere it is differentiable. All three activation functions I have presented so far have this property.

Logistic Functions A sigmoid function is defined as

$$\sigma(x) = \frac{1}{1+\exp(-x)},$$

and its derivative is

$$\sigma'(x) = \sigma(x)(1-\sigma(x)).$$

A hyperbolic tangent function is

$$\tanh(x) = \frac{\exp(2x)-1}{\exp(2x)+1},$$

[7] If anyone's interested in digging more into the whole field of automatic differentiation, try to Google it and you'll find tons of materials. One such reference is [4].

and its derivative is

$$\tanh'(x) = \left(\frac{2}{\exp(x) + \exp(-x)} \right)^2.$$

Piece-wise Linear Functions I described a rectified linear unit (rectifier or ReLU, [70, 41]) earlier:

$$\text{rect}(x) = \max(0, x).$$

It is clear that this function is not strictly differentiable, because of the discontinuity at $x = 0$. However, the chance of the input to this rectifier lands exactly at 0 has zero probability, meaning that we can forget about this extremely unlikely event. The derivative of the rectifier in this case is

$$\text{rect}'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases}$$

Although the rectifier has become the most widely used nonlinearity, especially, in deep learning's applications to computer vision,[8] there is a small issue with the rectifier. That is, for a half of the input space, the derivative is zero, meaning that the error (the output derivative from Eq. (3.15)) will be not well propagated through the rectifier function node.

In [42], the rectifier was extended to a maxout unit so as to avoid this issue of the existence of zero-derivative region in the input to the rectifier. The maxout unit of rank k is defined as

$$\text{maxout}(x_1, \ldots, x_k) = \max(x_1, \ldots, x_k),$$

and its derivative as

$$\frac{\partial \text{maxout}}{\partial x_i}(x_1, \ldots, x_k) = \begin{cases} 1, & \text{if } \max(x_1, \ldots, x_k) = x_i \\ 0, & \text{otherwise} \end{cases}$$

This means that the derivative is backpropagated only through one of the k inputs.

Stochastic Variables These activation functions work well with the backpropagation algorithm, because they are differentiable almost everywhere in the input space. However, what happens if a function is non-differentiable at all. One such example is a binary stochastic node, which is computed by

1. Compute $p = \sigma(x)$, where x is the input to the function node.
2. Consider p as a mean of a Bernoulli distribution, i.e., $\mathscr{B}(p)$.
3. Generate one sample $s \in \{0, 1\}$ from the Bernoulli distribution.
4. Output s.

[8] Almost all the winning entries in ImageNet Large Scale Visual Recognition Challenges (ILSVRC) use a convolutional neural network with rectifiers. See `http://image-net.org/challenges/LSVRC/`.

Clearly there is no derivative of this function node.

Does it mean that we're doomed in this case? Fortunately, no. Although I will not discuss about this any further in this course, Bengio et al. [6] provide an extensive list of approaches we can take in order to compute the derivative of the stochastic function nodes.

Chapter 4

Recurrent Neural Networks and Gated Recurrent Units

After the last lecture I hope that it has become clear how to build a multilayer perceptron. Of course, there are so many details that I did not mention, but are extremely important in practice. For instance, how many layers of simple transformations Eq. (3.8) should a multilayer perceptron have for a certain task? How wide (equiv. $\dim(\phi_0(\mathbf{x}))$) should each transformation be? What other transformation layers are there? What kind of learning rate η (see Eq. (2.5)) should we use? How should we schedule this learning rate over training? Answers to many of these questions are unfortunately heavily task-, data- and model-dependent, and I cannot provide any general answer to them.

4.1 Recurrent Neural Networks

Instead, I will move on to describing how we can build a neural network[1] to handle a variable length input. Until now the input x was assumed to be either a scalar or a vector of the fixed number of dimensions. From here on however, we remove this assumption of a fixed size input and consider the case of having a variable length input x.

What do I mean by a *variable length input*? A variable length input x is a *sequence* where each input x has a different number of elements. For instance, the first training example's input x^1 may consist of l^1 elements such that

$$x^1 = (x^1_1, x^1_2, \ldots, x^1_{l^1}).$$

Meanwhile, another example's input x^n may be a sequence of $l^n \neq l^1$ elements:

$$x^n = (x^n_1, x^n_2, \ldots, x^n_{l^n}).$$

Let's go back to very basic about dealing with these kinds of sequences. Furthermore, let us assume that each element x_i is binary, meaning that it is either 0 or 1. What

[1] Now, let me begin using a term neural network instead of a general function.

would be the most natural way to write a function that returns the number of 1's in an input sequence $x = (x_1, x_2, \dots, x_l)$? My answer is to first build a recursive function called ADD1, shown in Alg. 1. This function ADD1 will be called for each element of the input x, as in Alg. 2.

Algorithm 1 A function ADD1

$s \leftarrow 0$
function ADD1(v,s)
  if $v = 0$ **then return** s
  else return $s + 1$
  end if
end function

Algorithm 2 A function ADD1

$s \leftarrow 0$
for $i \leftarrow 1, 2, \dots, l$ **do** $s \leftarrow$ ADD1(x_i, s)
end for

There are two important components in this implementation. First, there is a memory s which counts the number of 1's in the input sequence x. Second, a single function ADD1 is applied to each symbol in the sequence *one at a time* together with the memory s. Thanks to these two properties, our implementation of the function ADD1 can be used with the input sequence of *any length*.

Now let us generalize this idea of having a memory and a recursive function that works over a variable length sequence. One likely most general case of this idea is a digital computer we use everyday. A computer program is a sequence x of instructions x_i. A central processing unit (CPU) reads each instruction of this program and manipulates its registers according to what the instruction says. Manipulating registers is often equivalent to manipulating any input–output (I/O) device attached to the CPU. Once one instruction is executed, the CPU moves on to the next instruction which will be executed with the content of the registers from the previous step. In other words, these registers work as a memory in this case (s from Alg. 2,) and the execution of an instruction by the CPU corresponds to a recursive function (ADD1 from Alg. 1.)

Both ADD1 and CPU are *hard coded* in the sense that they do what they have been designed and manufactured to do. Clearly, this is not what we want, because nobody knows how to design a CPU or a recursive function for natural language understanding, which is our ultimate goal. Instead what we want is to have a parametric recursive function that is able to read a sequence of (linguistic) symbols and use a memory in order to *understand natural languages*.

To build this parametric recursive function[2] that works on a variable-length input sequence $x = (x_1, x_2, \dots, x_l)$, we now know that there needs to be a memory. We will use one vector $\mathbf{h} \in \mathbb{R}^{d_h}$ as this memory vector. As is clear from Alg. 1, this recursive function takes as input both one input symbol x_t and the memory vector $\mathbf{h}$, and it

[2] In neural network research, we call this function a *recurrent neural network.*

returns the updated memory vector. It often helps to *time index* the memory vector as well, such that the input to this function is $\mathbf{h}_{t-1}$ (the memory after processing the previous symbol x_{t-1},) and we use $\mathbf{h}_t$ to denote the memory vector returned by the function. This function is then

$$h_t = f(x_t, \mathbf{h}_{t-1})$$

Now the big question is what kind of parametric form this recursive function f takes? We will follow the simple transformation layer from Eq. (3.8), in which case we get

$$f(x_t, \mathbf{h}_{t-1}) = g(\mathbf{W}\phi(x_t) + \mathbf{U}\mathbf{h}_{t-1}), \tag{4.1}$$

where $\phi(x_t)$ is a function that transforms the input symbol (often discrete) into a d-dimensional real-valued vector. $\mathbf{W} \in \mathbb{R}^{d_h \times d}$ and $\mathbf{U}^{d_h \times d_h}$ are parameters of this function. A nonlinear activation function g can be any function, but for now, we will assume that it is an element-wise nonlinear function such as tanh.

4.1.1 Fixed-Size Output y

Because our goal is to approximate an underlying, true function, we now need to think of how we use this recursive function to *return* an output y. As with the case of variable-length sequence input x, y can only be either a fixed-size output, such as a category to which the input x belongs, or a variable-length sequence output. Here let us discuss the case of having a fixed-size output y.

The most natural approach is to use the last memory vector $\mathbf{h}_l$ to produce the output (or more often output distribution.) Consider a task of binary classification where y is either positive (1) or negative (0), in which case a Bernoulli distribution fits perfectly. A Bernoulli distribution is fully characterized by a single parameter μ. Hence,

$$\mu = \sigma(\mathbf{v}^\top \mathbf{h}_l),$$

where $\mathbf{v} \in \mathbb{R}^{d_h}$ is a weight vector, and σ is a sigmoid function.

This now looks very much like the multilayer perceptron from Sec. 3.3. The whole function given an input sequence x computes

$$\mu = \sigma(\mathbf{v}^\top \underbrace{g(\mathbf{W}\phi(x_l) + \mathbf{U}g(\mathbf{W}\phi(x_{l-1}) + \mathbf{U}g(\mathbf{W}\phi(x_{l-2}) + \cdots g(\mathbf{W}\phi(x_1) + \mathbf{U}\mathbf{h}_0)\cdots))))}_{(a)\text{ recurrence}}), \tag{4.2}$$

where $\mathbf{h}_0$ is an initial memory state which can be simply set to an all-zero vector.

The main difference is that the input is not given only to the first simple transformation layer, but is given to all those transformation layers (one at a time.) Also, each transformation layer *shares* the parameters $\mathbf{W}$ and $\mathbf{U}$.[3] The first two steps of the

[3] Note that for brevity, I have omitted bias vectors. This should not matter much, as having a bias vector is equivalent to augmenting the input with a constant element whose value is fixed at 1. Why? Because,

$$[\mathbf{W}; \mathbf{b}] \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Note that as I have declared before all vectors are *column* vectors.

recurrence part (a) of Eq. (4.2) are shown as a computational graph in Fig. 4.1.

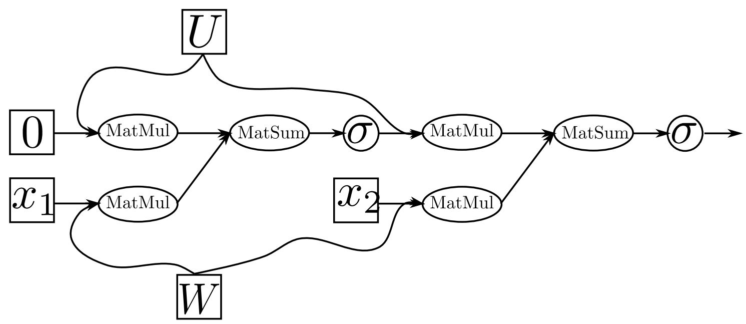


Figure 4.1: Sample computational graph of the recurrence in Eq. (4.2).

As this is not any special computational graph, the whole discussion on how to automate backpropagation (computing the gradient of the cost function w.r.t. the parameters) in Sec. 3.4 applies to recurrent neural networks directly, except for one potentially confusing point.

4.1.2 Multiple Child Nodes and Derivatives

It may be confusing how to handle those parameters that are shared across multiple time steps; **W** and **U** in Fig. 4.1. In fact, in the earlier section (Sec. 3.4), we did not discuss about what to do when the output of one node is fed into multiple function nodes. Mathematically saying, what do we do in the case of

$$c = g(f_1(x), f_2(x), \ldots, f_n(x))?$$

g can be any function, but let us look at two widely used cases:

- Addition: $g(f_1(x), \ldots, f_n(x)) = \sum_{i=1}^{n} f_i(x)$

$$\frac{\partial c}{\partial x} = \frac{\partial c}{\partial g} \sum_{i \in \{1,2,\ldots,n\}} \frac{\partial f_i}{\partial x}.$$

- Multiplication: $g(f_1(x), \ldots, f_n(x)) = \prod_{i=1}^{n} f_i(x)$

$$\frac{\partial c}{\partial x} = \frac{\partial c}{\partial g} \sum_{i \in \{1,2,\ldots,n\}} \left(\prod_{j \neq i} f_j(x) \right) \frac{\partial f_i}{\partial x}.$$

From these two cases, we can see that in general

$$\frac{\partial c}{\partial x} = \frac{\partial c}{\partial g} \sum_{i \in \{1,2,\ldots,n\}} \frac{\partial g}{\partial f_i} \frac{\partial f_i}{\partial x}.$$

This means that when multiple derivatives are *backpropagated* into a single node, the node should first *sum them* and multiply its summed derivative with its own derivative.

What does this mean for the shared parameters of the recurrent neural network? In an equation,

$$
\begin{aligned}
\frac{\partial C}{\partial W} = & \underbrace{\frac{\partial C}{\partial \mathrm{MatSum}^l}}_{(a)} \frac{\partial \mathrm{MatSum}^l}{\partial \mathrm{MatMul}^l} \frac{\partial \mathrm{MatMul}^l}{\partial W} \\
& + \underbrace{\frac{\partial C}{\partial \mathrm{MatSum}^l}}_{(a)} \underbrace{\frac{\partial \mathrm{MatSum}^l}{\partial \mathrm{MatSum}^{l-1}}}_{(b)} \frac{\partial \mathrm{MatSum}^{l-1}}{\partial \mathrm{MatMul}^{l-1}} \frac{\partial \mathrm{MatMul}^{l-1}}{\partial W} \\
& + \underbrace{\frac{\partial C}{\partial \mathrm{MatSum}^l}}_{(a)} \underbrace{\frac{\partial \mathrm{MatSum}^l}{\partial \mathrm{MatSum}^{l-1}}}_{(b)} \underbrace{\frac{\partial \mathrm{MatSum}^{l-1}}{\partial \mathrm{MatSum}^{l-2}}}_{(c)} \frac{\partial \mathrm{MatSum}^{l-2}}{\partial \mathrm{MatMul}^{l-2}} \frac{\partial \mathrm{MatMul}^{l-2}}{\partial W} \\
& + \cdots,
\end{aligned}
\tag{4.3}
$$

where the superscript l of each function node denotes the layer at which the function node resides.

Similarly to what we've observed in Sec. 3.4, many derivatives are shared across the terms inside the summation in Eq. (4.3). This allows us to compute the derivative of the cost function w.r.t. the parameter W efficiently by simply running the recurrent neural network backward.

4.1.3 Example: Sentiment Analysis

There is a task in natural language processing called *sentiment analysis*. As the name suggests, the goal of this task is to predict the sentiment of a given text. This is definitely one function that a human can do fairly well: when you read a critique's review of a movie, you can easily tell whether the critique likes, hates or is neutral to the movie. Also, even without a star rating of a product on Amazon, you can quite easily tell whether a user like it by reading her/his review of the product.

In this task, an input sequence x is a given text, and the fixed-size output is its label which is almost always one of positive, negative or neutral. Let us assume for now that the input is a *sequence of words*, where each word $\mathbf{x}_i$ is represented as a so-called one-hot vector.[4] In this case, we can use

$$\phi(x_t) = \mathbf{x}_t$$

in Eq. (4.1).

[4] A one-hot vector is a way to represent a discrete symbol as a binary vector. The one-hot vector $\mathbf{v}_i$ of a symbol $i \in V = \{1, 2, \ldots, |V|\}$ is

$$\mathbf{v}_i = [\underbrace{0, \ldots, 0}_{1,\ldots,i-1}, \underbrace{1}_{i}, \underbrace{0, \ldots, 0}_{i+1,\ldots,|V|}]^\top.$$

Once the input sequence, or paragraph in this specific example, is read, we get the last memory state $\mathbf{h}_l$ of the recurrent neural network. We will affine-transform $\mathbf{h}_l$ followed by the *softmax* function to obtain the conditional distribution of the output $y \in \{1,2,3\}$ (1: positive, 2: neutral and 3: negative):

$$\mu = [\mu_1, \mu_2, \mu_3]^\top = \mathrm{softmax}(\mathbf{V}\mathbf{h}_l),$$

where μ_1, μ_2 and μ_3 are the probabilities of "positive", "neural" and "negative". See Eq. (3.5) for more details on the softmax function.

Because this network returns a categorial distribution, it is natural to use the (categorical) cross entropy as the cost function. See Eq. (3.6). A working example of this sentiment analyzer based on recurrent neural networks will be introduced and discussed during the lab session.[5]

4.1.4 Variable-Length Output y: $|x| = |y|$

Let's generalize what we have discussed so far to recurrent neural networks here. Instead of a fixed-size output y, we will assume that the goal is to label each input symbol, resulting in the output sequence $y = (y_1, y_2, \ldots, y_l)$ of the same length as the input sequence x.

What kind of applications can you think of that returns the output sequence as long as the input sequence? One of the most widely studied problems in natural language processing is a problem of classifying each word in a sentence into one of part-of-speech tags, often called POS tagging (see Sec. 3.1 of [67].) Unfortunately, in my personal opinion, this is perhaps the least interesting problem of all time in natural language understanding, but perhaps the most well suited problem for this section.

In its simplest form, we can view this problem of POS tagging as classifying each word in a sentence as one of *noun*, *verb*, *adjective* and *others*. As an example, given the following input sentence x

$$x = (\mathrm{Children}, \mathrm{eat}, \mathrm{sweet}, \mathrm{candy}),$$

the goal is to output

$$y = (\mathrm{noun}, \mathrm{verb}, \mathrm{adjective}, \mathrm{noun}).$$

This task can be solved by a recurrent neural network from the preceding section (Sec. 4.1.1) after a quite trivial modification. Instead of waiting until the end of the sentence to get the last memory state of the recurrent neural network, we will use the *immediate memory state* to predict the label at each time step t.

At each time t, we get the immediate memory state $\mathbf{h}_t$ by

$$\mathbf{h}_t = f(x_t, \mathbf{h}_{t-1}), \tag{4.4}$$

where f is from Eq. (4.1). Instead of continuing on to processing the next word, we will first predict the label of the t-th input word x_t.

[5] For those eager to learn more, see `http://deeplearning.net/tutorial/lstm.html` in advance of the lab session.

This can be done by

$$\mu_t = [\mu_{t,1}, \mu_{t,2}, \mu_{t,3}, \mu_{t,4}]^\top = \text{softmax}(\mathbf{V}\mathbf{h}_t). \tag{4.5}$$

Four $\mu_{t,i}$'s correspond to the probabilities of the four categories; (1) noun, (2) verb, (3) adjective and (4) others.

From this output distribution at time step t, we can define a *per-step*, *per-sample* cost function:

$$C_{x,t}(\theta) = -\sum_{k=1}^{K} \mathbb{I}_{k=y}\mu_{t,k}, \tag{4.6}$$

where K is the number of categories, four in this case. We discussed earlier in Eq. (3.6). Naturally a per-sample cost function is defined as the sum of these per-step, per-sample cost functions:

$$C_x(\theta) = -\sum_{t=1}^{l}\sum_{k=1}^{K} \mathbb{I}_{k=y}\mu_{t,k}. \tag{4.7}$$

Incorporating the Output Structures This formulation of the cost function is equivalent to maximizing the log-probability of the correct output sequence given an input sequence, where the conditional log-probability is defined as

$$\log p(y|x) = \underbrace{\sum_{t=1}^{l} \underbrace{\log p(y_t|x_1,\ldots,x_t)}_{\text{Eq. (4.6)}}}_{\text{Eq. (4.7)}}. \tag{4.8}$$

This means that the network is predicting the label of the t-th input symbol using only the input symbols read up to that point (i.e., $x_1, x_2, \ldots, x_t$.)

In other words, this means that the recurrent neural network is *not* taking into account the structure of the output sequence. For instance, even without looking at the input sequence, in English it is well known that the probability of the next word being a noun increases if the current word is an adjective.[6] This kind of structures in the output are effectively ignored in this formulation.

Why is this so in this formulation? Because, we have made an assumption that the output symbols $y_1, y_2, \ldots, y_l$ are mutually independent conditioned on the input sequence. This is clear from Eq. (4.8) and the definition of the conditional independence:

$$\begin{aligned} & Y_1 \text{ and } Y_2 \text{ are conditionally independent dependent on } X \\ \iff & p(Y_1, Y_2|X) = p(Y_1|X)p(Y_2|x). \end{aligned}$$

If the underlying, true conditional distribution obeyed this assumption of conditional independence, there is no worry. However, this is a very strong assumption for

[6] Okay, this requires a more thorough analysis, but for the sake of the argument, which does not have to do anything with actual POS tags, let's believe that this is indeed the case.

many of the tasks we run into, apparently from the example of POS tagging. Then, how can we exploit the structure in the output sequence?

One simple way is to make a less strong assumption about the conditional probability of the output sequence y given x. For instance, we can assume that

$$\log p(y|x) = \sum_{i=1}^{l} \log p(y_i|y_{<i}, x_{\leq i}),$$

where $y_{<i}$ and $x_{\leq i}$ denote all the output symbols before the i-th one and all the input symbols up to the i-th one, respectively.

Now the question is how we can incorporate this into the existing formulation of a recurrent neural network from Eq. (4.4). It turned out that the answer is extremely simple. All we need to do is to compute the memory state of the recurrent neural network based not only on the current input symbol x_t and the previous memory state $\mathbf{h}_{t-1}$, but also on the previous output symbol y_{t-1} such that

$$\mathbf{h}_t = f(x_t, y_{t-1}, \mathbf{h}_{t-1}).$$

Similarly to Eq. (4.1), we can think of implementing f as

$$f(x_t, y_{t-1}, \mathbf{h}_{t-1}) = g(\mathbf{W}_x \phi_x(x_t) + \mathbf{W}_y \phi_y(y_{t-1}) + \mathbf{W}_h \mathbf{h}_{t-1}).$$

There are two questions naturally arising from this formulation. First, what do we do when computing $\mathbf{h}_1$? This is equivalent to saying what $\phi_y(y_0)$ is. There are two potential answers to this question:

1. Fix $\phi_y(y_0)$ to an all-zero vector
2. Consider $\phi_y(y_0)$ as an additional parameter

In the latter case, $\phi_y(y_0)$ will be estimated together with all the other parameters such as those weight matrices $\mathbf{W}_x$, $\mathbf{W}_y$, $\mathbf{W}_h$ and $\mathbf{V}$.

Inference The second question involves how to handle y_{t-1}. During training, it is quite straightforward, as our cost function (KL-divergence between the underlying, true distribution and the parametric conditional distribution $p(y|x)$, approximated by Monte Carlo method) says that we use the groundtruth value for y_{t-1}'s.

It is however not clear what we should do when we test the trained network, because then we are not given the groundtruth output sequence. This process of finding an output that maximizes the conditional (log-)probability is called *inference*[7]:

$$\hat{y} = \arg\max_y \log p(y|x)$$

[7] Okay, I confess. The term *inference* refers to a much larger class of problems, even if we consider only machine learning. However, let me simply use this term to refer to a task of finding the most likely output of a function.

The exact inference is quite straightforward. One can simply evaluate $\log p(y|x)$ for every possible output sequence and choose the one with the highest conditional probability. Unfortunately, this is almost always intractable, as the number of every possible output sequence grows exponentially with respect to the length of the sequence:

$$|\mathscr{Y}| = K^l,$$

where $\mathscr{Y}$, K and l are the set of all possible output sequences, the number of labels and the length of the sequence, respectively. Thus, this is necessary to resort to approximate search over the set $\mathscr{Y}$.

The most naive approach to approximate inference is a greedy one. With the trained model, you predict the first output symbol $\hat{y}_1$ based on the first input symbol x_1 by selecting the category of the highest probability $p(y_1|x_1)$. Now, given $\hat{y}_1$, x_1 and x_2, we compute $p(y_2|x_1, x_2, y_1)$ from which we select the next output symbol $\hat{y}_2$ with the highest probability. We continue this process iteratively until the last output symbol $\hat{y}_l$ is selected.

This is greedy in the sense that any early choice with a high conditional probability may turn out to be unlikely one due to extremely low conditional probabilities later on. It is highly related to the so-called *garden path sentence* problem. To know more about this, read, for instance, Sec. 3.2.4 of [67].

It is possible to alleviate this issue by considering $N < K$ best hypotheses of the output sequence at each time step. This procedure is called *beam search*, and we will discuss more about this in a later lecture on neural machine translation.

4.2 Gated Recurrent Units

4.2.1 Making Simple Recurrent Neural Networks *Realistic*

Let us get back to the analogy we made in Sec. 4.1. We compared a recurrent neural network to how CPU works. Executing a recurrent function f is equivalent to executing one of the instructions on CPU, and the memory state of the recurrent neural network is equivalent to the registers of the CPU. This analogy does sound plausible, except that it is not.

In fact, how a simple recurrent neural network works is far from being similar to how CPU works. I am now talking about how they are implemented in practice, but rather I'm talking at the conceptual level. What is it at the conceptual level that makes the simple recurrent neural network unrealistic?

An important observation we make about the simple recurrent neural network is that it *refreshes* the whole memory state at each time step. This is almost opposite to how the registers on a CPU are maintained. Each time an instruction is executed, the CPU does not clear up the whole registers and repopulate them. Rather, it works only on a small number of registers. All the other registers' values are stored as they were before the execution of the instruction.

Let's try to write this procedure mathematically. Each time, based on the choice of instruction to be executed, a subset of the registers of a CPU, or a subset of the elements in the memory state of a recurrent neural network, is selected. This can be written down as a binary vector $\mathbf{u} \in \{0,1\}^{n_h}$:

$$u_i = \begin{cases} 0, & \text{if the register's value does not change} \\ 1, & \text{if the register's value will change} \end{cases}$$

With this binary vector, which I will call an *update gate*, a new memory state or a new register value at time t can be computed as a convex interpolation such that

$$\mathbf{h}_t = (1 - \mathbf{u}) \odot \mathbf{h}_{t-1} + \mathbf{u} \odot \tilde{\mathbf{h}}_t, \tag{4.9}$$

where $\odot$ is as usual an element-wise multiplication. $\tilde{\mathbf{h}}_t$ denotes a new memory state or a new register value, after executing the instruction at time t.

Another unrealistic point about the simple recurrent neural network is that each execution considers the whole registers. It is almost impossible to imagine designing an instruction on a CPU that requires to read the values of all the registers. Instead, what almost always happens is that each instruction will consider only a small subset of the registers, which again we can use a binary vector to represent. Let me call it a *reset gate* $\mathbf{r} \in \{0,1\}^{n_h}$:

$$r_i = \begin{cases} 0, & \text{if the register's value will not be used} \\ 1, & \text{if the register's value will be used} \end{cases}$$

This reset gate can be multiplied to the register values *before* being used by the instruction at time t.[8] If we use a recursive function f from Eq. (4.1), it means that

$$\tilde{\mathbf{h}}_t = f(x_t, \mathbf{r} \odot \mathbf{h}_{t-1}) = g(\mathbf{W}\phi(x_t) + \mathbf{U}(\mathbf{r} \odot \mathbf{h}_{t-1})). \tag{4.10}$$

Now, let us put these two gates that are necessary to make the simple recurrent neural network more realistic into one piece. At each time step, the *candidate* memory state is computed based on a subset of the elements of the previous memory state:

$$\tilde{\mathbf{h}}_t = g(\mathbf{W}\phi(x_t) + \mathbf{U}(\mathbf{r} \odot \mathbf{h}_{t-1}))$$

A new memory state is computed as a linear interpolation between the previous memory state and this candidate memory state using the update gate:

$$\mathbf{h}_t = (1 - \mathbf{u}) \odot \mathbf{h}_{t-1} + \mathbf{u} \odot \tilde{\mathbf{h}}_t$$

See Fig. 4.2 for the graphical illustration.

4.2.2 Gated Recurrent Units

Now here goes a big question: *How are the update* **u** *and reset* **r** *gates computed?*

If we stick to our analogy to the CPU, those gates must be pre-configured *per instruction*. Those binary gates are dependent on the instruction. Again however, this

[8] It is important to note that this is *not* resetting the actual values of the registers, but only the input to the instruction/recursive function.

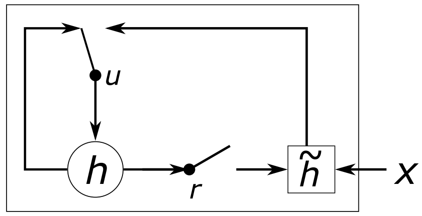


Figure 4.2: A graphical illustration of a gated recurrent unit [26].

is not what we want to do in our case. There is no set of predefined instructions, but the execution of any instruction corresponds to computing a recurrent *function* based on the input symbol and the memory state from the previous time step (see, e.g., Eq. (4.1).) Similarly to this what we want with the update and reset gates is that they are computed by a function which depends on the input symbol and the previous memory state.

This sounds like quite straightforward, except that we defined the gates to be binary. This means that whatever the function we use to compute those gates, the function will be a discontinuous function with zero derivative almost everywhere, except at the point where a sharp transition from 0 to 1 happens. We discussed the consequence of having an activation function with zero derivative almost everywhere in Sec. 3.4.1, and the conclusion was that it becomes very difficult to compute the gradient of the cost function efficiently *and* exactly with these discrete activation functions in a computational graph.

One simple solution which turned out to be extremely efficient is to consider those gates not as binary vectors but as real-valued coefficient vectors. In other words, we redefine the update and reset gates to be

$$\mathbf{u} \in [0,1]^{n_h}, \mathbf{r} \in [0,1]^{n_h}.$$

This approach makes these gates *leaky* in the sense that they always allow some leak of information through the gate.

In the case of the reset gate, rather than making a hard decision on which subset of the registers, or the elements of the memory state, will be used, it now decides *how much* information from the previous memory state will be used. The update gate on the other hand now controls how much content in the memory state will be replaced, which is equivalent to saying that it controls how much information will be *kept* from the previous memory state.

Under this definition we can simply use a sigmoid function from Eq. (3.11) to compute these gates:

$$\begin{aligned}
\mathbf{r} &= \sigma(\mathbf{W}_r \phi(x_t) + \mathbf{U}_r \mathbf{h}_{t-1}), \\
\mathbf{u} &= \sigma(\mathbf{W}_u \phi(x_t) + \mathbf{U}_u (\mathbf{r} \odot \mathbf{h}_{t-1})),
\end{aligned}$$

where $\mathbf{W}_r$, $\mathbf{U}_r$, $\mathbf{W}_u$ and $\mathbf{U}_u$ are the additional parameters.[9] Since the sigmoid function is differentiable everywhere, we can use the backpropagation algorithm (see Sec. 3.4)

[9] Note that this is not *the* formulation available for computing the reset and update gates. For instance,

to compute the derivatives of the cost function with respect to these parameters and estimate them together with all the other parameters.

We call this recurrent activation function with the reset and update gates a *gated recurrent unit* (GRU), and a recurrent neural network having this GRU as a gated recurrent network.

4.2.3 Long Short-Term Memory

The gated recurrent unit (GRU) is highly motivated by a much earlier work on long short-term memory (LSTM) units [47].[10] The LSTM was proposed in 1997 with the goal of building a recurrent neural network that can learn long-term dependencies across many number of timsteps, which was deemed to be difficult to do so with a simple recurrent neural network.

Unlike the element-wise nonlinearity of the simple recurrent neural network and the gated recurrent unit, the LSTM explicitly separates the memory state $\mathbf{c}_t$ and the output $\mathbf{h}_t$. The output is a small subset of the *hidden* memory state, and only this subset of the memory state is visibly *exposed* to any other part of the whole network.

How does a recurrent neural network with LSTM units decide how much of the memory state it will reveal? As perhaps obvious at this point, the LSTM uses a so-called *output* gate $\mathbf{o}$ to achieve this goal. Similarly to the reset and update gates of the GRU, the output gate is computed by

$$\mathbf{o} = \sigma(\mathbf{W}_o \phi(x_t) + \mathbf{U}_o \mathbf{h}_{t-1}).$$

This output vector is multiplied to the memory state $\mathbf{c}_t$ point-wise to result in the output:

$$\mathbf{h}_t = \mathbf{o} \odot \tanh(\mathbf{c}_t).$$

Updating the memory state $\mathbf{c}_t$ closely resembles how it is updated in the GRU (see Eq. (4.9).) A major difference is that instead of using a single update gate, the LSTM uses two gates, forget and input gates, such that

$$\mathbf{c}_t = \mathbf{f} \odot \mathbf{c}_{t-1} + \mathbf{i} \odot \tilde{\mathbf{c}}_t,$$

where $\mathbf{f} \in \mathbb{R}^{n_h}$, $\mathbf{i} \in \mathbb{R}^{n_h}$ and $\tilde{\mathbf{c}}_t$ are the forget gate, input gate and the candidate memory state, respectively.

The roles of those two gates are quite clear from their names. The forget gate decides how much information from the memory state will be *forgotten*, while the input gate controls how much informationa about the new input (consisting of the input

---

one can use the following definitions of the reset and update gates:

$$\mathbf{r} = \sigma(\mathbf{W}_r \phi(x_t) + \mathbf{U}_r \mathbf{h}_{t-1}),$$
$$\mathbf{u} = \sigma(\mathbf{W}_u \phi(x_t) + \mathbf{U}_u \mathbf{h}_{t-1}),$$

which is more parallelizable than the original formulation from [26]. This is because there is no more direct dependency between $\mathbf{r}$ and $\mathbf{u}$, which makes it possible to compute them in parallel.

[10] Okay, let me confess here. I was not well aware of long short-term memory when I was designing the gated recurrent unit together with Yoshua Bengio and Caglar Gulcehre in 2014.

symbol and the previous output) will be *inputted* to the memory. They are computed by

$$\mathbf{f} = \sigma(\mathbf{W}_f \phi(x_t) + \mathbf{U}_f \mathbf{h}_{t-1}), \qquad (4.11)$$
$$\mathbf{i} = \sigma(\mathbf{W}_i \phi(x_t) + \mathbf{U}_i \mathbf{h}_{t-1}).$$

The candidate memory state is computed similarly to how it was done with the GRU in Eq. (4.10):

$$\tilde{\mathbf{c}}_t = g(\mathbf{W}_c \phi(x_t) + \mathbf{U}_c \mathbf{h}_{t-1}), \qquad (4.12)$$

where g is often an element-wise tanh.

All the additional parameters specific to the LSTM–$\mathbf{W}_o, \mathbf{U}_o, \mathbf{W}_f, \mathbf{U}_f, \mathbf{W}_i, \mathbf{U}_i, \mathbf{W}_c$ and $\mathbf{U}_c$– are estimated together with all the other parameters. Again, every function inside the LSTM is differentiable everywhere, and we can use the backpropagation algorithm to efficient compute the gradient of the cost function with respect to all the parameters.

Although I have described one formulation of the long short-term memory unit here, there are many other variants proposed over more than a decade since it was first proposed. For instance, the forget gate in Eq. (4.11) was not present in the original work [47] but was fixed to 1. Gers et al. [40] proposed the forget gate few years after the LSTM was originally proposed, and it turned out to be one of the most crucial component in the LSTM. For more variants of the LSTM, I suggest you to read [43, 51].[11]

4.3 Why not Rectifiers?

4.3.1 Rectifiers Explode

Let us go back to the *simple recurrent neural network* which uses the simple transformation layer from Eq. (4.1):

$$f(x_t, \mathbf{h}_{t-1}) = g(\mathbf{W}\phi(x_t) + \mathbf{U}\mathbf{h}_{t-1}),$$

where g is an element-wise nonlinearity.

One of the most widely used nonlinearities is a hyperbolic tangent function tanh. This is unlike the case in feedforward neural networks (multilayer perceptrons) where a (unbounded) piecewise linear function, such as a rectifier and maxout, has become standard. In the case of feedforward neural networks, you can safely assume that everyone uses some kind of piecewise linear function as an activation function in the network. This has become pretty much standard since Krizhevsky et al. [57] shocked the (computer vision) research community by outperforming all the more traditional computer vision teams in the ImageNet Large Scale Visual Recognition Challenge 2012.[12]

[11] Interestingly, based on the observation in [51], it seems like the plain LSTM with a forget gate and the GRU seem to be close to the optimal gated unit we can find.

[12] `http://image-net.org/challenges/LSVRC/2012/results.html`

The main difference between logistic functions (tanh and sigmoid function) and piecewise linear functions (rectifiers and maxout) is that the former is bounded from both above and below, while the latter is bounded only from below (or in some cases, not bounded at all [44].[13])

This unbounded nature of piece-wise linear functions makes it difficult for them to be used in recurrent neural networks. Why is this so?

Let us consider the simplest case of unbounded element-wise nonlinearity; a linear function:

$$g(a) = a.$$

The hidden state after l symbols is

$$\begin{aligned}
\mathbf{h}_l =& \mathbf{U}(\mathbf{U}(\mathbf{U}(\mathbf{U}(\cdots) + \mathbf{W}\phi(x_{l-3})) + \mathbf{W}\phi(x_{l-2})) + \mathbf{W}\phi(x_{l-1})) + \mathbf{W}\phi(x_l) \\
=& \left(\prod_{l'=1}^{l-1} \mathbf{U}\right) \mathbf{W}\phi(x_1) + \left(\prod_{l'=1}^{l-2} \mathbf{U}\right) \mathbf{W}\phi(x_2) + \cdots + \mathbf{U}\mathbf{W}\phi(x_{l-1}) + \mathbf{W}\phi(x_l), \\
=& \sum_{t=1}^{l} \underbrace{\left(\prod_{l'=1}^{l-t} \mathbf{U}\right) \mathbf{W}\phi(x_t)}_{(a)}
\end{aligned} \tag{4.13}$$

where l is the length of the input sequence.

Let us assume that

- $\mathbf{U}$ is a full rank matrix
- The input sequence is sparse: $\sum_{t=1}^{l} \mathbb{I}_{\phi(x_t) \neq 0} = c$, where $c = O(1)$
- $[\mathbf{W}\phi(x)]_i > 0$ for all i

and consider Eq. (4.13) (a):

$$\mathbf{h}_l^{t'} = \left(\prod_{l'=1}^{l-t'} \mathbf{U}\right) \mathbf{W}\phi(x_{t'}). \tag{4.14}$$

Now, let's look at what happens to Eq. (4.14). First, the eigendecomposition of the matrix $\mathbf{U}$:

$$\mathbf{U} = \mathbf{Q}\mathbf{S}\mathbf{Q}^{-1},$$

where $\mathbf{S}$ is a diagonal matrix whose non-zero entries are eigenvalues. $\mathbf{Q}$ is an orthogonal matrix. Then

$$\prod_{l'=1}^{l-t'} \mathbf{U} = \mathbf{Q}\mathbf{S}^{l-t'}\mathbf{Q}^{-1},$$

[13] A parametric rectifier, or PReLU, is defined as

$$g(x) = \begin{cases} x, & \text{if } x \geq 0 \\ ax, & \text{otherwise} \end{cases},$$

where a is a parameter to be estimated together with all the other parameters of a network.

and

$$\left(\prod_{l'=1}^{l-t'} \mathbf{U}\right) \mathbf{W}\phi(x_{t'}) = \mathrm{diag}(\mathbf{S}^{l-t'}) \odot (\underbrace{\mathbf{Q}\mathbf{Q}^{-1}}_{=\mathbf{I}} \mathbf{W}\phi(x_{t'})),$$

where $\odot$ is an element-wise product.

What happens if the largest eigenvalue $e_{\max} = \max \mathrm{diag}(\mathbf{S})$ is larger than 1, the norm of $\mathbf{h}_l$ will *explode*, i.e., $\|\mathbf{h}_l\| \to \infty$. Furthermore, due to the assumption that $\mathbf{W}\phi(x_{t'}) > 0$, each element of $\mathbf{h}_l$ will explode to infinity as well. The rate of growth is exponentially with respect to the length of the input sequence, meaning that even when the input sequence is not too long, the norm of the memory state grows quickly if $e_{\max}$ is reasonably larger than 1.

This happens, because the nonlinearity g is *unbounded*. If g is bounded from both above and below, such as the case with tanh, the norm of the memory state is also bounded. In the case of $\tanh : \mathbb{R} \to [-1, 1]$,

$$\|\mathbf{h}_l\| \leq \dim(\mathbf{h}_l).$$

This is one reason why a logistic function, such as tanh and σ, is most widely used with recurrent neural networks, compared to piecewise linear functions.[14] I will call this recurrent neural network with tanh as an element-wise nonlinear function a *simple recurrent neural network*.

4.3.2 Is tanh a Blessing?

Now, the argument in the previous section may sound like tanh and σ are *the* nonlinear functions that one should use. This seems quite convincing for recurrent neural networks, and perhaps so for feedforward neural networks as well, if the network is *deep* enough.

Here let me try to convince you otherwise by looking at how the norm of backpropagated derivative behaves. Again, this is much easier to see if we assume the following:

- $\mathbf{U}$ is a full rank matrix
- The input sequence is sparse: $\sum_{t=1}^{l} \mathbb{I}_{\phi(x_t) \neq 0} = c$, where $c = O(1)$

Similarly to Eq. (4.13), let us consider a forward computational path until $\mathbf{h}_l$, however without assuming a linear activation function:

$$\mathbf{h}_l = g\left(\mathbf{U}g\left(\mathbf{U}g\left(\mathbf{U}g\left(\mathbf{U}(\cdots) + \mathbf{W}\phi\left(x_{l-3}\right)\right) + \mathbf{W}\phi\left(x_{l-2}\right)\right) + \mathbf{W}\phi\left(x_{l-1}\right)\right) + \mathbf{W}\phi\left(x_l\right)\right).$$

We will consider a subsequence of this process, in which all the input symbols are 0 except for the first symbol:

$$\mathbf{h}_{l_1} = g\left(\mathbf{U}g\left(\mathbf{U}\left(\cdots g\left(\mathbf{U}\mathbf{h}_{l_0} + \mathbf{W}\phi\left(x_{l_0+1}\right)\right)\right)\right)\right).$$

[14] However, it is not to say that piecewise linear functions are never used for recurrent neural networks. See, for instance, [59, 5].

It should be noted that as l approaches infinity, there will be at least one such subsequence whose length also approaches infinity due to the sparsity of the input we assumed.

From this equation, let's look at

$$\frac{\partial \mathbf{h}_{l_1}}{\partial \phi\left(x_{l_0+1}\right)}.$$

This measures the effect of the (l_0+1)-th input symbol x_{l_0+1} on the l_1-th memory state of the simple recurrent neural network. This is also the crucial derivative that needs to be computed in order to compute the gradient of the cost function using the automated backpropagation procedure described in Sec. 3.4.

This derivative can be rewritten as

$$\frac{\partial \mathbf{h}_{l_1}}{\partial \phi\left(x_{l_0+1}\right)} = \underbrace{\frac{\partial \mathbf{h}_{l_1}}{\partial \mathbf{h}_{l_0+1}}}_{(a)} \frac{\partial \mathbf{h}_{l_0+1}}{\partial \underline{\mathbf{h}}_{l_0+1}} \frac{\partial \underline{\mathbf{h}}_{l_0+1}}{\partial \phi\left(x_{l_0+1}\right)}.$$

Among these three terms in the left hand side, we will focus on the first one (a) which can be further expanded as

$$\frac{\partial \mathbf{h}_{l_1}}{\partial \mathbf{h}_{l_0+1}} = \left(\underbrace{\frac{\partial \mathbf{h}_{l_1}}{\partial \underline{\mathbf{h}}_{l_1}}}_{(b)} \underbrace{\frac{\partial \underline{\mathbf{h}}_{l_1}}{\partial \mathbf{h}_{l_1-1}}}_{(c)}\right) \left(\underbrace{\frac{\partial \mathbf{h}_{l_1-1}}{\partial \underline{\mathbf{h}}_{l_1-1}}}_{(b)} \underbrace{\frac{\partial \underline{\mathbf{h}}_{l_1-1}}{\partial \mathbf{h}_{l_1-2}}}_{(c)}\right) \cdots \left(\underbrace{\frac{\partial \mathbf{h}_{l_0+2}}{\partial \underline{\mathbf{h}}_{l_0+2}}}_{(b)} \underbrace{\frac{\partial \underline{\mathbf{h}}_{l_0+2}}{\partial \mathbf{h}_{l_0+1}}}_{(c)}\right). \tag{4.15}$$

Because this is a recurrent neural network, we can see that the analytical forms for the terms grouped by the parentheses in the above equation are identical except for the subscripts indicating the time index. In other words, we can simply only on one of those groups, and the resulting analytical form will be generally applicable to all the other groups.

First, we look at Eq. (4.15) (b), which is nothing but a derivative of a nonlinear activation function used in this simple recurrent neural network. The derivatives of the widely used logistic functions are

$$\begin{aligned}\sigma'(x) =& \sigma(x)(1-\sigma(x)),\\ \tanh'(x) =& 1-\tanh^2(x),\end{aligned}$$

as described earlier in Sec. 3.4.1. Both of these functions' derivatives are *bounded*:

$$0 < \sigma'(x) \leq 0.25, \tag{4.16}$$

$$0 < \tanh'(x) \leq 1. \tag{4.17}$$

In the simplest case in which g is a linear function (i.e., $x = g(x)$,) we do not even need to look at $\left\|\frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}\right\|$. We simply ignore all the $\frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}$ from Eq. (4.15).

Next, consider Eq. (4.15) (c). In this case of simple recurrent neural network, we notice that we have already learned how to compute this derivative earlier in Sec. 3.3.2:

$$\frac{\partial \underline{\mathbf{h}}_{t+1}}{\partial \mathbf{h}_t} = \mathbf{U}$$

From these two, we get

$$\begin{aligned}\frac{\partial \mathbf{h}_{l_1}}{\partial \mathbf{h}_{l_0+1}} &= \left(\frac{\partial \mathbf{h}_{l_1}}{\partial \underline{\mathbf{h}}_{l_1}}\mathbf{U}\right)\left(\frac{\partial \mathbf{h}_{l_1-1}}{\partial \underline{\mathbf{h}}_{l_1-1}}\mathbf{U}\right)\cdots\left(\frac{\partial \mathbf{h}_{l_0+2}}{\partial \underline{\mathbf{h}}_{l_0+2}}\mathbf{U}\right)\\ &= \prod_{t=l_0+2}^{l_1}\left(\frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}\mathbf{U}\right).\end{aligned}$$

Do you see how similar it looks like Eq. (4.14)? If the recurrent activation function f is linear, this whole term reduces to

$$\frac{\partial \mathbf{h}_{l_1}}{\partial \mathbf{h}_{l_0+1}} = \mathbf{U}^{l_1-l_0+1},$$

which according to Sec. 4.3.1, will explode as $l \to \infty$ if

$$e_{\max} > 1,$$

where $e_{\max}$ is the largest eigenvalue of $\mathbf{U}$. When $e_{\max} < 1$, it will *vanish*, i.e., $\|\frac{\partial \mathbf{h}_{l_1}}{\partial \mathbf{h}_{l_0+1}}\| \to 0$, exponentially fast.

What if the recurrent activation function f is *not* linear at all? Let's look at $\frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}\mathbf{U}$ as

$$\frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}\mathbf{U} = \underbrace{\begin{bmatrix} f_1' & 0 & \cdots & 0 \\ 0 & f_2' & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & f_{n_h}' \end{bmatrix}}_{=\mathrm{diag}\left(\frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}\right)} \left(\mathbf{Q}\mathbf{S}\mathbf{Q}^{-1}\right),$$

where we used the eigendecomposition of $\mathbf{U} = \mathbf{Q}\mathbf{S}\mathbf{Q}^{-1}$. This can be re-written into

$$\frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}\mathbf{U} = \mathbf{Q}\left(\mathrm{diag}\left(\frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}\right)\odot \mathbf{S}\right)\mathbf{Q}^{-1}.$$

This means that the eigenvalue of $\mathbf{U}$ will be scaled by the derivative of the recurrent activation function at each timestep. In this case, we can bound the maximum eigenvalue of $\frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}\mathbf{U}$ by

$$e^t_{\max} \leq \lambda e_{\max},$$

where λ is the upperbound on $g' = \frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}$. See Eqs. (4.16)–(4.17) for the upperbounds of the sigmoid and hyperbolic tangent functions.

In other words, if the largest eigenvalue of $\mathbf{U}$ is larger than $\frac{1}{\lambda}$, it is *likely* that this temporal derivative of $\mathbf{h}_{l_1}$ with respect to $\mathbf{h}_{l_0+1}$ will *explode*, meaning that its norm will grow exponentially large. In the opposite case of $e_{\max} < \frac{1}{\lambda}$, the norm of the temporal derivative likely shrinks toward 0. The former case is referred to as *exploding gradient*, and the latter *vanishing gradient*. These cases were studied already at the very early years of research in recurrent neural networks [8, 46].

Using tanh is a blessing in recurrent neural networks when running the network forward, as I described in the previous section. This is however not necessarily true in the case of backpropagating derivaties. Especially because, there is a higher chance of vanishing gradient with tanh, or even worse with σ. Why? Because $\frac{1}{\lambda} > 1$ for almost everywhere.

4.3.3 Are We Doomed?

Exploding Gradient Fortunately it turned out that the phenomenon of exploding gradient is quite easy to address. First, it is straightforward to detect whether the exploding gradient happened by inspecting the norm of the gradient fo the cost with respect to the parameters $\left\|\nabla_\theta \tilde{C}\right\|$. If the gradient's norm is larger than some predefined threhold $\tau > 0$, we can simply renormalize the norm of the gradient to be τ. Otherwise, we leave it as it is.

In mathematics,

$$\tilde{\nabla} = \begin{cases} \tau \frac{\nabla}{\|\nabla\|}, & \text{if } \|\nabla\| > \tau \\ \nabla, & \text{otherwise} \end{cases},$$

where we used the shorthand notiation ∇ for $\nabla_\theta \tilde{C}$. $\tilde{\nabla}$ is a rescaled gradient update direction which will be used by the stochastic gradient descent (SGD) algorithm from Sec. 2.2.2. This algorithm is referred to as *gradient clipping* [72].

Vanishing Gradient What about vanishing gradient? But, first, what does vanishing gradient mean? We need to understand the meaning of this phenomenon in order to tell whether this is a problem at all from the beginning.

Let us consider a case the variable-length output where $|x| = |y|$ from Sec. 4.1.4. Let's assume that there exists a clear dependency between the output label y_t and the input symbol $x_{t'}$, where $t' \ll t$. This means that the empirical cost will decrease when the weights are adjusted such that

$$\log p(y_t = y_t^* | \ldots, \phi(x_{t'}), \ldots)$$

is maximized, where y_t^* is the ground truth output label at time t. The value of $\phi(x_{t'})$ has great influence on the t-th output y_t, and the influence can be measured by

$$\frac{\partial \log p(y_t = y_t^* | \ldots)}{\partial \phi(x_{t'})}.$$

Instead of exactly computing $\frac{\partial \log p(y_t = y_t^* | \ldots)}{\partial \phi(x_{t'})}$, we can approximate it by the finite difference method. Let $\varepsilon \in \mathbb{R}^{\dim(\phi(x_{t'}))}$ be a vector of which each element is a very small real value ($\varepsilon \approx 0$.) Then,

$$\frac{\partial \log p(y_t = y_t^* | \ldots)}{\partial \phi(x_{t'})} = \lim_{\varepsilon \to 0} (\log p(y_t = y_t^* | \ldots, \phi(x_{t'}) + \varepsilon, \ldots) \\ - \log p(y_t = y_t^* | \ldots, \phi(x_{t'}), \ldots,)) \oslash \varepsilon,$$

where $\oslash$ is an element-wise division. This shows that $\frac{\partial \log p(y_t = y_t^* | \ldots)}{\partial \phi(x_{t'})}$ computes the difference in the t-th output probability with respect to the change in the value of the t'-th input.

In other words $\frac{\partial \log p(y_t = y_t^* | \ldots)}{\partial \phi(x_{t'})}$ directly reflects the degree to which the t-th output y_t *depends* on the t'-th input $x_{t'}$, *according to the network*. To put it in another way, $\frac{\partial \log p(y_t = y_t^* | \ldots)}{\partial \phi(x_{t'})}$ reflects how much dependency the *recurrent neural network has captured the dependency between y_t and $x_{t'}$*.

Let's rewrite

$$\frac{\partial \log p(y_t = y_t^* | \ldots)}{\partial \phi(x_{t'})} = \frac{\partial \log p(y_t = y_t^* | \ldots)}{\partial \mathbf{h}_t} \underbrace{\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \cdots \frac{\partial \mathbf{h}_{t'+1}}{\partial \mathbf{h}_{t'}}}_{(a)} \frac{\partial \mathbf{h}_{t'}}{\partial \phi(x_t)}.$$

The terms marked with (a) looks exactly identical to Eq. (4.15). We have already seen that this term can easily *vanish* toward zero with a high probability (see Sec. 4.3.2.)

This means that the recurrent neural network is unlikely to capture this dependency. This is especially true when the (temporal) distance between the output and input, i.e., $|t - t'| \gg 0$.

The biggest issue with this vanishing behaviour is that there is no straightforward way to avoid it. We cannot tell whether $\frac{\partial \log p(y_t = y_t^* | \ldots)}{\partial \phi(x_{t'})} \approx 0$ is due to the lack of this dependency in the true, underlying function or due to the wrong configuration (parameter setting) of the recurrent neural network. If we are certain that there are indeed these long-term dependencies, we may simultaneously minimize the following auxiliary term together with the cost function:

$$\sum_{t=1}^{T} \left(1 - \frac{\left\| \frac{\partial \tilde{C}}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t} \right\|}{\left\| \frac{\partial \tilde{C}}{\partial \mathbf{h}_{t+1}} \right\|} \right)^2.$$

This term, which was introduced in [72], is minimized when the norm of the derivative does not change as it is being backpropagated, effectively forcing the gradient *not* to vanish.

This term however was found to help significantly only when the target task, or the underlying function, does indeed exhibit long-term dependencies. How can we know in advance? Pascanu et al. [72] showed this with the well-known toy tasks which were specifically designed to exhibit long-term dependencies [46].

4.3.4 Gated Recurrent Units Address Vanishing Gradient

Will the same problems of vanishing gradient happen with the gated recurrent units (GRU) or the long short-term memory units (LSTM)? Let us write the memory state at time t:

$$\begin{aligned}
\mathbf{h}_t =& \mathbf{u}_t \odot \tilde{\mathbf{h}}_t + (1-\mathbf{u}_t) \odot \left(\mathbf{u}_{t-1} \odot \tilde{\mathbf{h}}_{t-1} + (1-\mathbf{u}_{t-1}) \odot \left(\mathbf{u}_{t-2} \odot \tilde{\mathbf{h}}_{t-2} + (1-\mathbf{u}_{t-2}) \odot (\cdots)\right)\right) \\
=& \mathbf{u}_t \odot \tilde{\mathbf{h}}_t + (1-\mathbf{u}_t) \odot \mathbf{u}_{t-1} \odot \tilde{\mathbf{h}}_{t-1} + (1-\mathbf{u}_t) \odot (1-\mathbf{u}_{t-1}) \odot \mathbf{u}_{t-2} \odot \tilde{\mathbf{h}}_{t-2} + \cdots
\end{aligned}$$

Let's be more specific and see what happens to this with respect to $x_{t'}$:

$$\begin{aligned}
\mathbf{h}_t =& \mathbf{u}_t \odot \tilde{\mathbf{h}}_t + (1-\mathbf{u}_t) \odot \mathbf{u}_{t-1} \odot \tilde{\mathbf{h}}_{t-1} + (1-\mathbf{u}_t) \odot (1-\mathbf{u}_{t-1}) \odot \mathbf{u}_{t-2} \odot \tilde{\mathbf{h}}_{t-2} + \cdots \\
&+ \underbrace{\left(\prod_{k=t,\dots,t'+1} (1-\mathbf{u}_k)\right) \odot \mathbf{u}_{t'}}_{(a)} \odot \tanh\left(\mathbf{W}\phi(x_{t'}) + \mathbf{U}(\mathbf{r}_{t'} \odot \mathbf{h}_{t'-1})\right),
\end{aligned} \tag{4.18}$$

where $\prod$ is for element-wise multiplication.

What this implies is that the GRU effectively introduces a *shortcut* from time t' to t. The change in $x_{t'}$ will directly influence the value of $\mathbf{h}_t$, and subsequently the t-th output symbol y_t. In other words, all the issue with the simple recurrent neural network we discussed earlier in Sec. 4.3.3.

The update gate controls the strength of these shortcuts. Let's assume for now that the update gate is fixed to some predefined value between 0 and 1. This effectively makes the GRU a leaky integration unit [5]. However, as it is perhaps clear from Eq. (4.18) that we will inevitably run into an issue. Why is this so?

Let's say we are sure that there are many long-term dependencies in the data. It is natural to choose a large coefficient for the leaky integration unit, meaning the update gate is close to 1. This will definitely help carrying the dependency across many time steps, but this inevitably carries *unnecessary* information as well. This means that much of the representational power of the output function $g_{\text{out}}(\mathbf{h}_t)$ is wasted in *ignoring* those unnecessary information.

If the update gate is fixed to something substantially smaller than 1, all the shortcuts (see Eq. (4.18) (a)) will exponentially vanish. Why? Because it is a repeated multiplication of a scalar small than 1. In other words, it does not really help to have a leaky integration unit in the place of a simple tanh unit.

This is however not the case with the actual GRU or LSTM, because those update gates are *not* fixed but are adaptive with respect to the input. If the network detects that there is an important dependency being captured, the update gate will be closed ($u_j \approx 0$.) This will effectively strengthen the shortcut connection (see Eq. (4.18) (a).) When the network detects that there is no dependency anymore, it will open the update gate ($u_j \approx 1$), which effectively cuts off the shortcut. How does the network know, or detect, the existence or lack of these dependencies? Do we need to manually code this up? I will leave these questions for you to figure out.

Chapter 5

Neural Language Models

5.1 Language Modeling: First Step

What does it mean for a machine to understand natural language? In other words, how can we tell that the machine understood natural language? These are the two equivalent questions that are at the core of this course.

One of the most basic capability of a machine that can signal us that it indeed understands natural language is for the machine to tell how likely a given sentence is. Of course, this is extremely ill-defined, as we probably cannot define the *likeliness* of a sentence, because there are many different types of unlikeliness. For instance, a sentence "Colorless green ideas sleep furiously" from Chomsky's [29] is *unlikely* according to our common sense, because

1. An object ("idea") cannot be both "colorless" and "green."
2. An object cannot "sleep" "furiously."
3. An "idea" does not "sleep."

On the other hand, this sentence is a grammatically correct sentence.

Let's take a look at another sentence "Jane and me went to see a movie yesterday." Grammatically, this is not the most correct sentence one can make. It should be "Jane and I went to see a movie yesterday." Even with a grammatical error in the original sentence however, the meaning of the sentence is clear to me, and perhaps is much more understandable than the sentence "colorless green ideas sleep furiously." Furthermore, many people likely say this (saying "me" instead of "I") quite often. This sentence is thus likely according to our common sense, but is not likely according to the grammar.

This observation makes us wonder what is the criterion to use. Is it correct for a machine to tell whether the sentence is likely by analyzing its grammatical correctness? Or, is it possible that the machine should deem a sentence likely only when its meaning agrees well with common sense regardless of its grammatical correctness in the most strict sense?

As we discussed in the first lecture of the course, we are more interested in approaching natural language as a means for one to communicate ideas to a listener. In this sense, language use is a function which takes as input the surrounding environment, including the others' speech, and returns linguistic response, and this function is *not* given but learned via observing others' use of language and the reinforcement by the surrounding environment [83]. Also, throughout this course, we are not concerned too much about the existing syntactic, or grammatical, structures underlying natural language, which makes it difficult for us to say anything about the grammatical correctness of a given sentence.

In short, we take the route here that the likeliness of a sentence be determined based on its agreement with common sense. The common sense here is captured by everyday use of natural language, which consequently implies that the statistics of natural language use can be a strong indicator for determining the likely of a natural language sentence.

5.1.1 What if those linguistic structures do exist

Of course, as we discussed earlier in Sec. 1.1 and in this section, not everyone agrees. This is due to the fact that a perfect grammatical sentence may be considered unlikely, just because it does not happen often. In other words, statistical approaches to *language modeling* may conclude that a sentence with perfectly valid grammatical construction is unlikely. Is this a problem?

This problem of telling how likely a given sentence is can be viewed very naturally as building a probabilistic model of sentences. In other words, given a sentence S, what is the probability $p(S)$ of S? Let us briefly talk about what this means for the case of viewing the likeliness of a sentence as equivalent to its grammatical correctness.[1]

We first assume that there is an underlying linguistic structure G which has generated the observed sentence S. Of course, we do not know the correct G in advance, and unfortunately no one will tell us what the correct G is.[2] Thus, G is a hidden variable in this case. This hidden structure G generates the observed sentence S according to an unknown conditional distribution $p(S|G)$. Each and every grammatical structure G is assigned a prior probability which is also unknown in advance.[3]

With the conditional distribution $S|G$ and the prior distribution G, we easily get the joint distribution S,G by

$$p(S,G) = p(S|G)p(G),$$

from the definition of conditional probability.[4] From this joint distribution we get the

[1] Why briefly and why here? Because, we will not pursue this line at all after this section.

[2] Here, the correct G means the G that generated S, not the whole structure of G which is assumed to exist according to a certain set of rules.

[3] This is not necessarily true. If we believe that each and every grammatical correct sentence is equally likely and that each correct grammatical structure generates a single corresponding sentence, the prior distribution over the hidden linguistic structure is such that any correct structure is given an equal probability while any incorrect structure is given a zero probability. But, of course, if we think about it, there are clearly certain structures that are more prevalent and others that are not.

[4] A conditional probability of A given B is defined as

$$p(A|B) = \frac{p(A,B)}{p(B)}$$

distribution over a given sentence S by marginalizing out G:

$$p(S) = \sum_G p(S,G).$$

This means that we should compute how likely a given sentence S is with respect to all possible underlying linguistic structure. This is very likely intractable, because there must be infinite possible such structures.

Instead of computing $p(S)$ exactly we can simply look at its lowerbound. For instance, one simplest, and probably not the best, way to do so is

$$p(S) = \sum_G p(S,G) \geq p(S,\hat{G}),$$

where $\hat{G} = \arg\max_G p(S,G) = \arg\max_G p(G|S)$.[5]

This lowerbound is tight, i.e., $p(S) = p(S,\hat{G})$, when there is only a single true underlying linguistic structure $\hat{G}$ given S. What this says is that there is no other possible linguistic structure possible for a single observed sentence, i.e., no ambiguity in inferring the correct linguistic structure. In other words, we can compute the probability or likeliness of a given sentence by inferring its correct underlying linguistic structure.

However, there are a few issues here. First, it is not clear which formalism G follows, and we have briefly discussed about this at the very beginning of this course. Second, it is quite well known that most of the formalisms do indeed have uncertainty in inference. Again, we looked at one particular example in Sec. 1.1.2. These two issues make many people, including myself, quite uneasy about this type of *model-based* approaches.

In the remaining of this chapter, I will thus talk about model-free approaches (as opposed to these model-based approaches.)

5.1.2 Quick Note on Linguistic Units

Before continuing, there is one question that must be bugging you, or at least has bugged me a lot: *what is the minimal linguistic unit?*

If we think about written text, the minimal unit does seem like a character. With spoken language, the minimal unit seems to be a phoneme. But, is this the level at which we want to model the process of *understanding* natural language? In fact, to most of the existing natural language processing researchers as well as some (or most) linguists, the answer to this question is a hard "no."

The main reason is that these low-level units, both characters and phonemes, do not convey any meaning themselves. Does a Latin alphabet "q" have its own meaning? The answer by most of the people will be "no." Then, starting from this alphabet "q", how far should we climb up in the hierarchy of linguistic units to reach a level at which the unit begins to convey its own meaning? "qu" does not seem to have its own meaning still. "qui" in French means "who", but in English it does not really say much. "quit" in English is a valid word that has its own meaning, and similarly "quiet" is a valid word that has its own meaning, quite apart from that of "quit."

[5] This inequality holds due to the definition of probability, which states that $p(X) \geq 0$ and $\sum_X p(X) = 1$.

It looks like a word is the level at which meaning begins to form itself. However, this raises a follow-up question on the definition of a word: *What is a word?*

It is tempting to say that a sequence of non-blank characters is a word. This makes everyone's life so much easier, because we can simply split each sentence by a blank space to get a sequence of words. Unfortunately this is a very bad strategy. The simplest counter example to this definition of words is a token (which I will use to refer to a sequence of non-blank characters) consisting of a word followed by a punctuation. If we simply split a sentence into words by blank spaces, we will get a bunch of redundant words. For instance, "llama", "llama,", "llama.", "llama?", ""llama", "llama"" and "llama!" will all be distinct words. We will run into an issue of exploding vocabulary with any morphologically rich language. Furthermore, in some languages such as Chinese, there is no blank space at all inside a sentence, in which case this simple strategy will completely fail to give us any meaningful, small linguistic unit other than sentences.

Now at this point it almost seems like the best strategy is to use each character as a linguistic unit. This is not necessarily true due to the highly nonlinear nature of orthography.[6] There are many examples in which this nonlinear nature shows its difficulty. One such example is to consider the following three words: "quite", "quiet" and "quit".[7] All three character sequences have near identical forms, but their corresponding meanings differ from each other substantially. In other words, any function that maps from these character sequences to the corresponding meanings will have to be extremely nonlinear and thus difficult to be learned from data. Of course, this is an area with active research, and I hope I am not giving you an impression that characters are not the units to use (see, e.g., [52].)

Now then the question is whether there is some middle ground between characters and words (or blank-space-separated tokens) that are more suitable to be used as elementary linguistic units (see, e.g., [82].) Unfortunately this is again an area with active research. Hopefully, we will have time later in the course to discuss this issue further. For now, we will simply use blank-space-separated tokens as our linguistic units.

5.2 Statistical Language Modeling

Regardless of which linguistic unit we use, any natural language sentence S can be represented as a sequence of T discrete symbols such that

$$S = (w_1, w_2, \dots, w_T).$$

Each symbol is one element from a *vocabulary* V which contains all possible symbols:

$$V = \left\{v_1, v_2, \dots, v_{|V|}\right\},$$

where $|V|$ is used to mean the size of the vocabulary, or the number of all symbols.

[6] Orthography is defined as "the study of spelling and how letters combine to represent sounds and form words."

[7] I would like to thank Bart van Merrienboer for this example.

The problem of language modeling is equivalent to finding a model that assigns a probability $p(S)$ to a sentence:

$$p(S) = p(w_1, w_2, \ldots, w_T). \tag{5.1}$$

Of course, we are not given this distribution and need to learn this from data.

Let's say we are given data D which contains N sentences such that

$$D = \left\{S^1, S^2, \ldots, S^N\right\},$$

where each sentence S^n is

$$S^n = (w_1^n, w_2^n, \ldots, w_{T^n}^n),$$

meaning that each sentence has a different length.

Given this data D, let us estimate the probability of a certain sentence S. This is quite straightforward:

$$p(S) = \frac{\sum_{n=1}^{N} \mathbb{I}_{S=S^n}}{N}, \tag{5.2}$$

where $\mathbb{I}$ is the indicator function defined earlier in Eq. (3.7) which is defined as

$$\mathbb{I}_{S=S^n} = \begin{cases} 1, & \text{if } S = S^n \\ 0, & \text{otherwise} \end{cases}$$

This is equivalent to counting how many times S occurs in the data.[8]

5.2.1 Data Sparsity/Scarcity

Has this solved the whole problem of language model? No, unfortunately not. The very major issue here is that however large your corpus is, it is unlikely to contain all reasonable sentences in the world. Let's do simple counting here.

There are $|V|$ symbols in a vocabulary. Each sentence can be as long as T symbols. Then, there are $|V|^T$ possible sentences. A reasonable range for the sentence length T is roughly between 1 to 50, meaning that there are

$$\sum_{T=1}^{50} |V|^T$$

possible sentences. As it's quite clear, this is a huge space of sentences.

Of course, not all those sentences are plausible. This is however conceivable that even the fraction of that space will be gigantic, especially considering that the size of vocabulary often goes up to 100k to 1M words. Many of the plausible sentences will not appear in the corpus. Is this true? In fact, yes, it is.

It is quite easy to find such an example. For instance, Google Books Ngram Viewer[9] lets you search for a sentence or a sequence of up to five English words from

[8] A data set consisting of (written) text is often referred to as a *corpus*.

[9] `https://books.google.com/ngrams`

Figure 5.1: A picture of a llama lying down. From `https://en.wikipedia.org/wiki/Llama`

the gigantic corpus of Google Books. Let me try to search for a very plausible sentence "I like llama," and the Google Books Ngram[10] Viewer returns an error saying that "Ngrams not found: I like llama." (see Fig. 5.1 in the case you are not familiar with a llama.) See Fig. 5.2 as an evidence.

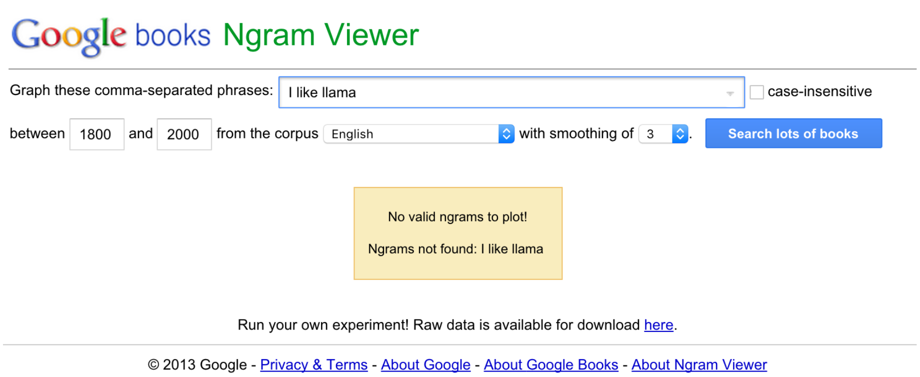


Figure 5.2: A resulting page of Google Books Ngram Viewer for the query "I like llama".

What does this mean for the estimate in Eq. (5.2)? It means that this estimator will be too harsh for many of the plausible sentences that do not occur in the data. As soon as a given sentence does not appear *exactly* as it is in the corpus, this estimator will say that there is a *zero* probability of the given sentence. Although the sentence "I like llama" is a likely sentence, according to this estimator in Eq. (5.2), it will be deemed extremely unlikely.

This problem is due to the issue of *data sparsity*. Data sparsity here refers to the

[10] We will discuss what Ngrams are in the later sections.

phenomenon where a training set does not cover the whole space of input sufficiently. In more concrete terms, most of the points in the input space, which have non-zero probabilities according to the true, underlying distribution, do not appear in the training set. If the size of a training set is assumed to be fixed, the severity of data sparsity increases as the average, or maximum length of the sentences. This follows from the fact that the size of the input space, the set of all possible sentences, grows with respect to the maximum possible length of a sentence.

In the next section, we will discuss the most straightforward approach to addressing this issue of data sparsity.

5.3 n-Gram Language Model

The fact that the issue of data sparsity worsens as the maximum length of sentences grows hints us a straightforward approach to addressing this: *limit the maximum length of phrases/sentences we estimate a probability on*. This idea is a foundation on which a so-called n-gram language model is based.

In the n-gram language model, we first rewrite the probability of a given sentence S from Eq. (5.1) into

$$p(S) = p(w_1, w_2, \ldots, w_T) = p(w_1)p(w_2|w_1)\cdots \underbrace{p(w_k|w_{<k})}_{(a)} \cdots p(w_T|w_{<T}), \tag{5.3}$$

where $w_{<k}$ denotes all the symbols before the k-th symbol w_k. From this, the n-gram language model makes an important assumption that each conditional probability (Eq. (5.3) (a)) is only conditioned on the $n-1$ preceding symbols only, meaning

$$p(w_k|w_{<k}) \approx p(w_k|w_{k-n}, w_{k-n+1}, \ldots, w_{k-1}).$$

This results in

$$p(S) \approx \prod_{t=1}^{T} p(w_t|w_{t-n}, \ldots, w_{t-1}).$$

What does this mean? Under this assumption we are saying that any symbol in a sentence is *predictable* based on the $n-1$ preceding symbols. This is in fact a quite reasonable assumption in many languages. For instance, let us consider a phrase “I am from”. Even without any more context information surrounding this phrase, such as surrounding words and the identity of a speaker, we know that the word following this phrase will be likely a name of place or country. In other words, the probability of a name of place or country given the three preceding words “I am from” is higher than that of any other words.

But, of course, this assumption does not always work. For instance, consider a phrase “In Korea, more than half of all the residents speak $\underbrace{\text{Korean}}_{(a)}$.” Let us focus on the last word “Korean” (marked with (a).) We immediately see that it will be useful to condition its conditional probability on the second word “Korea”. Why is this so?

Because the conditional probability of "Korean" following "speak" should significantly increase over all the other words (that correspond to other languages) knowing the fact that the sentence is talking about the residents of "Korea". This requires the conditional distribution to be conditioned on at least 10 words ("," is considered a separate word,) and this certainly will not be captured by n-gram language model with $n < 9$.

From these examples it is clear that there's a natural trade-off between the quality of probability estimate and statistical efficiency based on the choice of n in n-gram language modeling. The higher n the longer context the conditional distribution has, leading to a better model/estimate (second example,) however resulting in a situation of more sever data sparsity (see Sec. 5.2.1.) On the other hand, the lower n leads to the worse language modeling (second example), but this will avoid the issue of data sparsity.

***n*-gram Probability Estimation** We can estimate the n-gram conditional probability $p(w_k|w_{k-n},\dots,w_{k-1})$ from the training corpus. Since it is a *conditional* probability, we need to rewrite it according to the definition of the conditional probability:

$$p(w_k|w_{k-n},\dots,w_{k-1}) = \frac{p(w_{k-n},\dots,w_{k-1},w_k)}{p(w_{k-n},\dots,w_{k-1})} \tag{5.4}$$

This rewrite implies that the n-gram probability is equivalent to counting the occurrences of the n-gram $(w_{k-n},\dots,w_k)$ among all n-grams starting with $(w_{k-n},\dots,w_{k-1})$.

Let us consider the denominator first. The denominator can be computed by the marginalizing the k-th word (w' below):

$$p(w_{k-n},\dots,w_{k-1}) = \sum_{w' \in V} p(w_{k-n},\dots,w_{k-1},w'). \tag{5.5}$$

From Eq. (5.2), we know how to estimate $p(w_{k-n},\dots,w_{k-1},w')$:

$$p(w_{k-n},\dots,w_{k-1},w') \approx \frac{c(w_{k-n},\dots,w_{k-1},w')}{N_n}, \tag{5.6}$$

where $c(\cdot)$ is the number of occurrences of the given n-gram in the training corpus, and N_n is the number of all n-grams in the training corpus.

Now let's plug Eq. (5.6) into Eqs. (5.4)–(5.5):

$$p(w_k|w_{k-n},\dots,w_{k-1}) = \frac{\cancel{\frac{1}{N_n}} c(w_{k-n},\dots,w_{k-1},w_k)}{\cancel{\frac{1}{N_n}} \sum_{w' \in V} c(w_{k-n},\dots,w_{k-1},w')} \tag{5.7}$$

5.3.1 Smoothing and Back-Off

Note that I am missing many references this section, as I am writing this on my travel. I will fill in missing references once I'm back from my travel.

The biggest issue of having an n-gram that never occurs in the training corpus is that any sentence containing the n-gram will be given a zero probability regardless of how likely all the other n-grams are. Let us continue with the example of "I like llama". With an n-gram language model built using all the books in Google Books, the following, totally valid sentence[11] will be given a zero probability:

- "I like llama which is a domesticated South American camelid.[12]

Why is this so? Because the probability of this sentence is given as a product of all possible trigrams:

$$
\begin{aligned}
&p(\text{"I", "like", "llama", "which", "is", "a", "domesticated", "South", "American", "camelid"}) \\
=&p(\text{"I"})p(\text{"like"}|\text{"I"})\underbrace{p(\text{"llama"}|\text{"I","like"})}_{=0}\cdots p(\text{"camelid"}|\text{"South","American"}) \\
=&0
\end{aligned}
$$

One may mistakenly believe that we can simply increase the size of corpus (collecting even more data) to avoid this issue. However, remember that "*data sparsity is almost always an issue in statistical modeling*" [22], which means that more data call for better statistical models with often more parameters leading to the issue of data sparsity.

One way to alleviate this problem is to assign a small probability to all *unseen* n-grams. At least, in this case, we will assign some small, non-zero probability to any sentence, thereby avoiding a valid, but zero-probability sentence under the n-gram language model. One simplest implementation of this approach is to assume that each and every n-gram occurs at least α times and any occurrence in the training corpus is in addition to this background occurrence.

In this case, the estimate of an n-gram becomes

$$
\begin{aligned}
p(w_k|w_{k-n},\dots,w_{k-1}) =&\frac{\alpha + c(w_{k-n},w_{k-n+1},\dots,w_k)}{\sum_{w'\in V}(\alpha + c(w_{k-n},w_{k-n+1},\dots,w'))} \\
=&\frac{\alpha + c(w_{k-n},w_{k-n+1},\dots,w_k)}{\alpha|V| + \sum_{w'\in V} c(w_{k-n},w_{k-n+1},\dots,w')},
\end{aligned}
$$

where $c(w_{k-n},w_{k-n+1},\dots,w_k)$ is the number of occurrences of the given n-gram in the training corpus. $c(w_{k-n},w_{k-n+1},\dots,w')$ is the number of occurrences of the given n-gram if the last word w_k is substituted with a word w' from the vocabulary V. α is often set to be a scalar such that $0 < \alpha \leq 1$. See the difference from the original estimate in Eq. (5.7).

It is quite easy to see that this is a quite horrible estimator: how does it make sense to say that every unseen n-gram occurs with the same frequency? Also, knowing that this is a horrible approach, what can we do about this?

One possibility is to smooth the n-gram probability by interpolating between the estimate of the n-gram probability in Eq. (5.7) and the estimate of the $(n-1)$-gram probability. This can written down as

$$
\begin{aligned}
p^S(w_k|w_{k-n},\dots,w_{k-1}) =&\lambda(w_{k-n},\dots,w_{k-1})p(w_k|w_{k-n},\dots,w_{k-1}) \\
&+ (1-\lambda(w_{k-n},\dots,w_{k-1}))p^S(w_k|w_{k-n+1},\dots,w_{k-1}). \quad (5.8)
\end{aligned}
$$

[11] This is not strictly true, as I should put "a" in front of the llama.

[12] The description of a llama taken from Wikipedia: `https://en.wikipedia.org/wiki/Llama`

This implies that the n-gram (smoothed) probability is computed recursively by the lower-order n-gram probabilities. This is clearly an effective strategy, considering that falling off to the lower-order n-grams contains at least some information of the original n-gram, unlike the previous approach of adding a scalar α to every possible n-gram.

Now a big question here is how the interpolation coefficient λ is computed. The simplest approach we can think of is to fit it to the data as well. However, the situation is not that easy, as using the same training corpus, which was used to estimate $p(w_k|w_{k-n},\dots,w_{k-1})$ according to Eq. (5.7), will lead to a degenerate case. What is this degenerate case? If the same corpus is used to fit both the non-smoothed n-gram probability and λ's, the optimal solution is to simply set all λ's to 1, as that will assign the high probabilities to all the n-grams. Therefore, one needs to use a separate corpus to fit λ's.

More generally, we may rewrite Eq. (5.8) as

$$p^S(w_k|w_{k-n},\dots,w_{k-1}) = \begin{cases} \alpha(w_k|w_{k-n},\dots,w_{k-1}), \text{ if } c(w_{k-n},\dots,w_{k-1},w_k) > 0 \\ \gamma(w_{k-n+1},\dots,w_k)p^S(w_k|w_{k-n+1},\dots,w_{k-1}), \text{ otherwise} \end{cases} \tag{5.9}$$

following the notation introduced in [53]. Specific choices of α and γ lead to a number of different smoothing techniques. For an extensive list of these smoothing techniques, see [22].

Before ending this section on smoothing techniques for n-gram language modeling, let me briefly describe one of the most widely used smoothing technique, called the modified Kneser-Ney smoothing (KN smoothing), described in [22]. This modified KN smoothing is efficiently implemented in the open-source software package called KenLM [45].

First, let us define some quantities. We will use n_k to denote the total number of n-grams that occur exactly k times in the training corpus. With this, we define the following so-called discounting factors:

$$\begin{aligned} Y &= \frac{n_1}{n_1 + 2n_2} \\ D_1 &= 1 - 2Y\frac{n_2}{n_1} \\ D_2 &= 2 - 3Y\frac{n_3}{n_2} \\ D_{3+} &= 3 - 4Y\frac{n_4}{n_3}. \end{aligned}$$

Also, let us define the following quantities describing the number of all possible words following a given n-gram with a specified frequency l:

$$N_l(w_{k-n},\dots,w_{k-1}) = |\{c(w_{k-n},\dots,w_{k-1},w_k) = l\}|$$

The modified KN smoothing then defines α in Eq. (5.9) to be

$$\alpha(w_k|w_{k-n},\dots,w_{k-1}) = \frac{c(w_{k-n},\dots,w_{k-1},w_k) - D(c(w_{k-n},\dots,w_{k-1},w_k))}{\sum_{w'\in V} c(w_{k-n},\dots,w_{k-1},w')},$$

where D is

$$D(c) = \begin{cases} 0, & \text{if } c = 0 \\ D_1, & \text{if } c = 1 \\ D_2, & \text{if } c = 2 \\ D_{3+}, & \text{if } c \geq 3 \end{cases}$$

And, γ is defined as

$$\gamma(w_{k-n}, \dots, w_{k-1}) = \frac{D_1 N_1(w_{k-n}, \dots, w_{k-1}) + D_2 N_2(w_{k-n}, \dots, w_{k-1}) + D_{3+} N_{3+}(w_{k-n}, \dots, w_{k-1})}{\sum_{w' \in V} c(w_{k-n}, \dots, w_{k-1}, w')}.$$

For details on how this modified KN smoothing has been designed, see [22].

5.3.2 Lack of Generalization

Although n-gram language modelling works like a charm in many cases. This is still not totally satisfactory, because of the lack of *generalization*. What do I mean by *generalization* here?

Consider an example where three trigrams[13] were observed from a training corpus: "chases a cat", "chases a dog" and "chases a rabbit". There is a clear pattern here. The pattern is that it is highly likely that "chases a" will be followed by an animal.

How do we know this? This is a trivial example of humans' generalization ability. We have noticed a higher-level concept, in this case an animal, from observing words such as "cat", "dog" and "rabbit", and based on this concept, we generalize this knowledge (that "chases a" is followed by an animal) to unseen trigrams in the form of "chases a [animal]".

This however does not happen with n-gram language model. As an example, let's consider a trigram "chases a llama". Unless this specific trigram occurred more than once in the training corpus, the conditional probability given by n-gram language modeling will be zero.[14] This issue is closely related to data sparsity, but the main difference is that it is not the lack of data, or n-grams, but the lack of world knowledge. In other words, there *exist* relevant n-grams in the training corpus, but n-gram language modelling is not able to exploit these.

At this point, it almost seems trivial to address this issue by incorporating existing knowledge into language modelling. For instance, one can think of using a dictionary to find the definition of a word in interest (continuing on from the previous example, the definition of "llama") and letting the language model notice that "llama" is a "*a*

[13] Is "trigram" a proper term? Certainly not, but it is widely accepted by the whole community of natural language processing researchers. Here's an interesting discussion on how n-grams should be referred to as, from [67]: "*these alternatives are usually referred to as a bigram, a trigram, and a four-gram model, respectively. Revealing this will surely be enough to cause any Classicists who are reading this book to stop, and to leave the field to uneducated engineering sorts ... with the declining levels of education in recent decades ... some people do make an attempt at appearing educated by saying quadgram* "

[14] Here we assume that no smoothing or backoff is used. However, even when these techniques are used, we cannot be satisfied, since the probability assigned to this trigram will be at best reasonable up to the point that the n-gram language model is giving as high probability as the bigram "chases a". In other words, we do not get any generalization based on the fact that a "llama" is an animal similar to a "cat", "dog" or "rabbit".

domesticated pack animal of the camel family found in the Andes, valued for its soft woolly fleece." Based on this, the language model should figure out that the probability of "chases a llama" should be similar to "chases a cat", "chases a dog" or "chases a rabbit" because all "cat", "dog" and "rabbit" are animals according to the dictionary.

This is however not satisfactory for us. First, those definitions are yet another natural language text, and letting the model understand it becomes equivalent to natural language understanding (which is the end-goal of this whole course!) Second, a dictionary or any human-curated knowledge base is an inherently limited resource. These are limited in the sense that they are often static (not changing rapidly to reflect the changes in language use) and are often too generic, potentially not capturing any domain-specific knowledge.

In the next section, I will describe an approach purely based on statistics of natural language that is able to alleviate this lack of generalization.

5.4 Neural Language Model

One thing we notice from n-gram language modelling is that this boils down to computing the conditional distribution of a next word w_k given $n-1$ preceding words $w_{k-n}, \dots, w_{k-1}$. In other words, the goal of n-gram language modeling is to find a function that takes as input $n-1$ words and returns a conditional probability of a next word:

$$p(w_k | w_{k-n}, \dots, w_{k-1}) = f_\theta^{w_k}(w_{k-n}, \dots, w_{k-1}).$$

This is almost exactly what we have learned in Chapter 2.

First, we should define the input to this language modelling function. Clearly the input will be a sequence of $n-1$ words, but the question is how each of these words will be represented. Since our goal is to put the least amount of prior knowledge, we want to represent each word such that each and every word in the vocabulary is equi-distant away from the others. One encoding scheme that achieves this goal is 1-of-K coding.

In this 1-of-K coding scheme, each word i in the vocabulary V is represented as a binary vector $\mathbf{w}_i$ whose sum equals 1. To denote the i-th word with the vector $\mathbf{w}_i$, we set the i-th element of the vector $\mathbf{w}_i$ to be 1 (and consequently all the other elements are set to zero.) Mathematically,

$$\mathbf{w}_i = [0, 0, \dots, \underbrace{1}_{i\text{-th element}}, \dots, 0]^\top \in \{0, 1\}^{|V|} \tag{5.10}$$

This kind of vector is often called a one-hot vector.

It is easy to see that this encoding scheme perfectly fits our goal of having minimal prior, because

$$|\mathbf{w}_i - \mathbf{w}_j| = \begin{cases} 1, & \text{if } i \neq j \\ 0, & \text{otherwise} \end{cases}$$

Now the input to our function is a sequence of $n-1$ such vectors, which I will denote by $(\mathbf{w}^1, \mathbf{w}^2, \ldots, \mathbf{w}^{n-1})$. As we will use a neural network as a function approximator here,[15] these vectors will be multiplied with a weight matrix $\mathbf{E}$. After this, we get a sequence of continuous vectors $(\mathbf{p}^1, \mathbf{p}^2, \ldots, \mathbf{p}^{n-1})$, where

$$\mathbf{p}^j = \mathbf{E}^\top \mathbf{w}^j$$

and $\mathbf{E} \in \mathbb{R}^{|V| \times d}$.

Before continuing to build this function, let us see what it means to multiply the transpose of a matrix with an one-hot vector from left. Since only one of the elements of the one-hot vector is non-zero, all the rows of the matrix will be ignored except for the row corresponding to the index of the non-zero element of the one-hot vector. This row is multiplied by 1, which simply gives us the same row as the result of this whole matrix–vector multiplication. In short, the multiplication of the transpose of a matrix with an one-hot vector is equivalent to *slicing out a single row from the matrix*.

In other words, let

$$\mathbf{E} = \left[\begin{array}{c} \mathbf{e}_1 \\ \mathbf{e}_2 \\ \vdots \\ \mathbf{e}_{|V|} \end{array} \right], \tag{5.11}$$

where $\mathbf{e}_i \in \mathbb{R}^d$. Then,

$$\mathbf{E}^\top \mathbf{w}_i = \mathbf{e}_i.$$

This view has two consequences. First, in practice, it will be much more efficient computationally to implement this multiplication as a simple table look-up. For instance, in Python with NumPy, do

```
p = E[i,:]
```

instead of

```
p = numpy.dot(E.T, w_i)
```

Second, from this perspective, we can see each row of the matrix $\mathbf{E}$ as a continuous-space representation of a corresponding word. $\mathbf{e}_i$ will be a vector representation of the i-th word in the vocabulary V. This representation is often called a *word embedding* and should reflect the underlying meaning of the word. We will discuss this further shortly.

Closely following [7], we will simply concatenate the continuous-space representations of the input words such that

$$\mathbf{p} = \left[\mathbf{p}^1; \mathbf{p}^2; \ldots; \mathbf{p}^{n-1}\right]^\top$$

[15] Obviously, this does not have to be true, but at the end of the day, it is unclear if there is any parametric function approximation other than neural networks.

This vector $\mathbf{p}$ is a representation of $n-1$ input words in a continuous vector space and often referred to as a *context vector*.

This context vector is fed through a composition of nonlinear feature extraction layers. We can for instance apply the simple transformation layer from Eq. (3.8) such that

$$\mathbf{h} = \tanh(\mathbf{W}\mathbf{p} + \mathbf{b}), \tag{5.12}$$

where $\mathbf{W}$ and $\mathbf{b}$ are the parameters.

Once a set of nonlinear layers has been applied to the context vector, it's time to compute the output probability distribution. In this case of language modelling, the distribution outputted by the function is a categorical distribution. We discussed how we can build a function to return a categorical distribution already in Sec. 3.1.2.

As a recap, a categorical distribution defines a probability of one event happening among K discrete events. The probability of the k-th event happening is often denoted as μ_k, and

$$\sum_{k=1}^{K} \mu_k = 1.$$

Therefore, the function needs to return a K-dimensional vector $[\mu_1, \mu_2, \ldots, \mu_K]$. In this case of language modelling, $K = |V|$ and μ_i corresponds to the probability of the i-th word in the vocabulary for the next word.

As discussed earlier in Sec. 3.1.2, we can use softmax to compute each of those output probabilities:

$$p(w_n = k | w_1, w_2, \ldots, w_{n-1}) = \mu_k = \frac{\exp(\mathbf{u}_k^\top \mathbf{h} + c_k)}{\sum_{k'=1}^{|V|} \exp(\mathbf{u}_{k'}^\top \mathbf{h} + c_{k'})}, \tag{5.13}$$

where $\mathbf{u}_k \in \mathbb{R}^{\dim(\mathbf{h})}$.

This whole function is called a neural language model. See Fig. 5.3 (a) for the graphical illustration of neural language model.

5.4.1 How does Neural Language Model Generalize to Unseen n-Grams? – Distributional Hypothesis

Now that we have described neural language model, let us take a look into what happens inside. Especially, we will focus on how the model generalizes to unseen n-grams.

The previously described neural language model can be thought of as a composite of two function $(g \circ f)$. The first stage f projects a sequence of context words, or preceding $n-1$ words to a continuous vector space:

$$f : \{0,1\}^{|V| \times n-1} \to \mathbb{R}^d$$

We will call the resulting vector $\mathbf{h}$ a *context vector*. The second stage g maps this continuous vector $\mathbf{h}$ to the target word probability, by applying affine transformation to the vector $\mathbf{h}$ followed by softmax normalization.

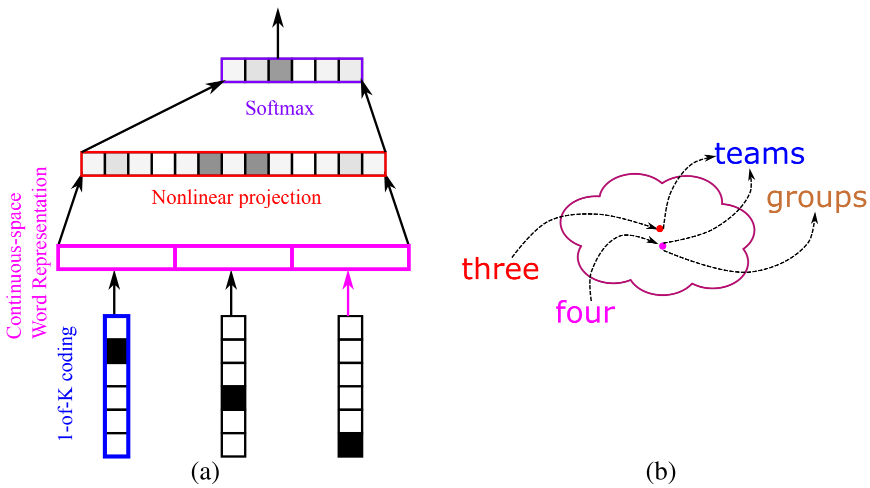


Figure 5.3: (a) Schematics of neural language model. (b) Example of how neural language model generalizes to an unseen n-gram.

Let us look more closely at what g does in Eq. (5.13). If we ignore the effect of the bias c_k for now, we can clearly see that the probability of the k-th word in the vocabulary is large when the output vector $\mathbf{u}_k$ (or the k-th row of the output matrix $\mathbf{U}$) is well aligned with the context vector $\mathbf{h}$. In other words, the probability of the next word being the k-th word in the vocabulary is roughly proportional to the inner product between the context vector $\mathbf{h}$ and the corresponding target word vector $\mathbf{u}_k$.

Now let us consider two context vectors $\mathbf{h}_j$ and $\mathbf{h}_k$. These contexts are followed by a similar set of words, meaning that the conditional distributions of the next word are similar to each other. Although these distributions are defined over all possibility target words, let us look at the probabilities of only one of the target words w_l:

$$p_j^l = p(w_l|\mathbf{h}_j) = \frac{1}{Z_j} \exp\left(\mathbf{w}_l^\top \mathbf{h}_j\right),$$
$$p_k^l = p(w_l|\mathbf{h}_k) = \frac{1}{Z_k} \exp\left(\mathbf{w}_l^\top \mathbf{h}_k\right).$$

The ratio between p_j^l and p_k^l is then[16]

$$\frac{p_j^l}{p_k^l} = \frac{Z_k}{Z_j} \exp\left(\mathbf{w}_l^\top (\mathbf{h}_j - \mathbf{h}_k)\right).$$

From this, we can clearly see that in order for the ratio $\frac{p_j^l}{p_k^l}$ to be 1, i.e., $p_j^l = p_k^l$,

$$\mathbf{w}_l^\top (\mathbf{h}_j - \mathbf{h}_k) = 0. \tag{5.14}$$

[16] Note that both p_j^l and p_k^l are positive due to our use of *softmax*.

Now let us assume that $\mathbf{w}_l$ is not an all-zero vector, as otherwise it will be too dull a case. In this case, the way to achieve the equality in Eq. (5.14) is to drive the context vectors $\mathbf{h}_j$ and $\mathbf{h}_k$ to each other. In other words, the context vectors must be similar to each other (in terms of Euclidean distance) in order to result in similar conditional distributions of the next word.

What does this mean? This means that the neural language model must project $(n-1)$-grams that are followed by the same word to nearby points in the context vector space, while keeping the other n-grams away from that neighbourhood. This is necessary in order to give a similar probability to the same word. If two $(n-1)$-grams, which are followed by the same word in the training corpus, are projected to far away points in the context vector space, it naturally follows from this argument that the probability over the next word will differ substantially, resulting in a bad language model.

Let us consider an extreme example, where we do bigram modeling with the training corpus comprising only three sentences:

- There are **three teams** left for the qualification.
- **four teams** have passed the first round.
- **four groups** are playing in the field.

We will focus on the bold-faced phrases; "three teams", "four teams" and "four group". The first word of each of these bigrams is a context word, and neural language model is asked to compute the probability of the word following the context word.

It is important to notice that neural language model *must* project "three" and "four" to nearby points in the context space (see Eq. (5.12).) This is because the context vectors from these two words need to give a similar probability to the word "teams". This naturally follows from our discussion earlier on how dot product preserves the ordering in the space. And, from these two context vectors (which are close to each other), the model assigns similar probabilities to "teams" and "groups", because they occur in the training corpus. In other words, the target word vector $\mathbf{u}_{\text{teams}}$ and $\mathbf{u}_{\text{groups}}$ will also be similar to each other, because otherwise the probability of "teams" given "four" ($p(\text{teams}|\text{four})$) and "groups" given "four" ($p(\text{groups}|\text{four})$) will be very different despite the fact that they occurred equally likely in the training corpus.

Now, let's assume the case where we use the neural language model trained on this tiny training corpus to assign a probability to an unseen bigram "three groups". The neural language model will project the context word "three" to a point in the context space close to the point of "four". From this context vector, the neural language model will *have to* assign a high probability to the word "groups", because the context vector $\mathbf{h}_{\text{three}}$ and the target word vector $\mathbf{u}_{\text{groups}}$ well align. Thereby, even without ever seeing the bigram "three groups", the neural language model can assign a reasonable probability. See Fig. 5.3 (b) for graphical illustration.

What this example shows is that neural language model automatically learns the similarity among different context words (via context vectors $\mathbf{h}$), and also among different target words (via target word vectors $\mathbf{u}_k$), by exploiting *co-occurrences* of words. In this example, the neural language model learned that "four" and "three" are similar from the fact that both of them occur together with "teams". Similarly, in the target side, the neural language model was able to capture the similarity between "teams" and "groups" by noticing that they both follow a common word "four".

This is a clear, real-world demonstration of the so-called distributional hypothesis. Distributional hypothesis states that "words which are similar in meaning appear in similar distributional contexts" [36]. By observing which words a given word co-occurs together, it is possible to peek into the word's underlying meaning. Of course, this is only a partial picture[17] into the underlying meaning of each word, or as a matter of fact a phrase, but surely still a very interesting property that is being naturally exploited by neural language model.

In neural language model, the most direct way to observe the effect of this distributional hypothesis/structure is to investigate the first layer's weight matrix $\mathbf{E}$ in Eq. (5.11). This weight matrix can be considered as a set of dense vectors of the words in the input vocabulary $\left\{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_{|V|}\right\}$, and any visualization technique, such as principal component analysis (PCA) or t-SNE [89], can be used to project each high-dimensional word vector into a lower-dimensional space (often 2-D or 3-D).

5.4.2 Continuous Bag-of-Words Language Model: Maximum Pseudo–Likelihood Approach

This is about time someone asks a question why we are only considering the *preceding* words when doing language modelling. Is it a good assumption that the conditional distribution over a word is only dependent on preceding words?

In fact, we do not have to do so. We can certainly model a natural language sentence such that each word in a sentence is conditioned on $2n$ surrounding words (n words to the left and n words to the right.) In this case, we get a *Markov random field (MRF) language model* [50].

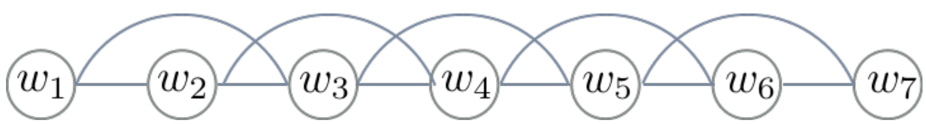


Figure 5.4: An example Markov random field language model (MRF-LM) with the order $n = 1$.

In a Markov random field (MRF) language model (MRF-LM), we say each word in a given sentence is a random variable w_i. We connect each word with its $2n$ surrounding words with *undirected* edges, and these edges represent the conditional dependency structure of the whole MRF-LM. An example of an MRF-LM with $n = 1$ is shown in Fig. 5.4.

A probability over a Markov random field is defined as a product of clique potentials. A potential is defined for each clique as a positive function whose input is the values of the random variables in the clique. In the case of MRF-LM, we will assign 1 as a potential to every clique except for cliques of two random variables (in

[17] We will discuss why this is only a partial picture later on.

other words, we only use pairwise potentials only.) The pairwise potential between the words i and j is defined as

$$\phi(\mathbf{w}^i, \mathbf{w}^j) = \exp\left((\mathbf{E}^\top \mathbf{w}^i)^\top \mathbf{E}^\top \mathbf{w}^j\right) = \exp\left(\mathbf{e}_{w^i}^\top \mathbf{e}_{w^j}\right),$$

where $\mathbf{E}$ is from Eq. (5.11), and $\mathbf{w}^i$ is the one-hot vector of the i-th word. One must note that this is one possible implementation of the pairwise potential, and there may be other possibilities, such as to replace the dot product between the word vectors ($\mathbf{e}_{w_i}^\top \mathbf{e}_{w_j}$) with a deeper network.

With this pairwise potential, the probability over the whole sentence is defined as

$$p(w_1, w_2, \ldots, w_T) = \frac{1}{Z} \prod_{t=1}^{T-n} \prod_{j=t}^{t+n} \phi(\mathbf{w}^t, \mathbf{w}^j) = \frac{1}{Z} \exp\left(\sum_{t=1}^{T-n} \mathbf{e}_{w^t}^\top \mathbf{e}_{w^j}\right),$$

where Z is the normalization constant. This normalization constant makes the product of the potentials to be a probability and often is at the core of computational intractability in Markov random fields.

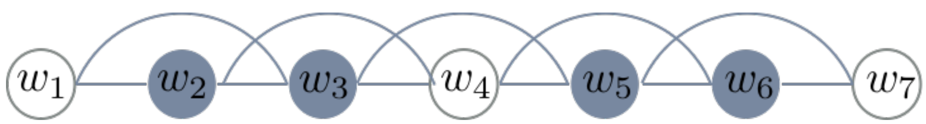


Figure 5.5: Gray nodes indicate the Markov blank of the fourth word.

Although compute the full sentence probability is intractable in this MRF-LM, it is quite straightforward to compute the conditional probability of each word w^i given all the other words. When computing the conditional probability, we must first notice that the conditional probability of w^i *only* depends on the values of other words included in its *Markov blanket.* In the case of Markov random fields, the Markov blanket of a random variable is defined as a set of all *immediate neighbours*, and it implies that the conditional probability of w^i is dependent only on n preceding words and the n following words. See Fig. 5.5 for an example.

Keeping this in mind, we can easily see that

$$p(w^i | w^{i-n}, \ldots, w^{i-1}, w^{i+1}, \ldots, w^{i+n}) = \frac{1}{Z'} \exp\left(\mathbf{e}_{w^i}^\top \left(\sum_{k=1}^{n} \mathbf{e}_{w^{i-k}} + \sum_{k=1}^{n} \mathbf{e}_{w^{i+k}}\right)\right),$$

where Z' is a normalization constant computed by

$$Z' = \sum_{v \in V} \exp\left(\mathbf{e}_v^\top \left(\sum_{k=1}^{n} \mathbf{e}_{w^{i-k}} + \sum_{k=1}^{n} \mathbf{e}_{w^{i+k}}\right)\right).$$

Do you see a stark similarity to neural language model we discussed earlier? This conditional probability is a shallow neural network with a single *linear* hidden layer

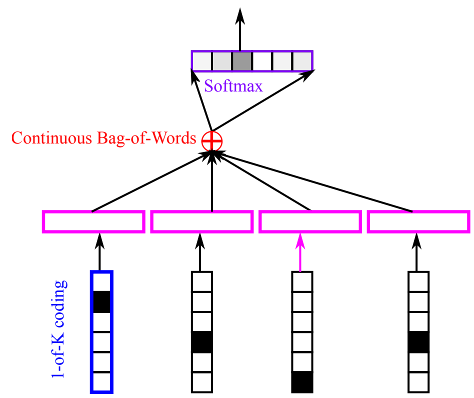


Figure 5.6: Continuous Bag-of-Words model approximates the conditional distribution over the j-th word w_j under the MRF-LM.

whose input are the context words (n preceding and n following words) and the output is the conditional distribution of the center word w_i. We will talk about this shortly in more depth. See Fig. 5.6 for graphical illustration.

Now we know that it is often difficult to compute the full sentence probability $p(w^1, \ldots, w^T)$ due to the intractable normalization constant Z. We however know how to compute the conditional probabilities (for all words) quite tractably. The former fact implies that it is perhaps not the best idea to maximize log-likelihood to train this model.[18] The latter however sheds a bit of light, because we can train a model to maximize pseudo–likelihood [10] instead.[19]

Pseudo–likelihood of the MRF-LM is defined as

$$\log \mathrm{PL} = \sum_{i=1}^{T} \log p(w^i | w^{i-n}, \ldots, w^{i-1}, w^{i+1}, \ldots, w^{i+n}). \tag{5.15}$$

Maximizing this pseudo–likelihood is equivalent to training a neural network in Fig. 5.6 which approximates each conditional distribution $p(w^i | w^{i-n}, \ldots, w^{i-1}, w^{i+1}, \ldots, w^{i+n})$ to give a higher probability to the ground-truth center word in the training corpus.

Unfortunately, even after training the model by maximizing the pseudo–likelihood in Eq. (5.15), we do not have a good way to compute the full sentence probability under this model. Under certain conditions maximizing pseudo–likelihood indeed converges to the maximum likelihood solution, but this does not mean that we can use the product of all the conditionals as a replacement of the full sentence probability. However, this does not mean that we cannot use this MRF-LM as a language model, since given a fixed model, the pseudo–probability (the product of all the conditionals) can score different sentences.

[18] However this is not to say maximum likelihood in this case is impossible. There are different ways to approximate the full sentence probability under this model. See [50] for one such approach.

[19] See the note by Amir Globerson (later modified by David Sontag) available at `http://cs.nyu.edu/~dsontag/courses/inference14/slides/pseudolikelihood_notes.pdf`.

This is in contrast to the neural language model we discussed earlier in Sec. 5.4. In the case of neural language model, we were able to compute the probability of a given sentence by computing the conditional probability of each word, reading from left until the end of the sentence. This is perhaps one of the reasons why the MRF-LM is not often used in practice as a language model. Then, you must ask why I even bothered to explain this MRF-LM in the first place.

This approach, which was proposed in [68] as a continuous bag-of-words (CBoW) model,[20] was found to exhibit an interesting property. That is, the word embedding matrix $\mathbf{E}$ learned as a part of this CBoW model very well reflects underlying structures of words, and this has become one of the darling models by natural language processing researchers in recent years. We will discuss further in the next section.

Skip-Gram and Implicit Matrix Factorization In [68], another model, called skip-gram, is proposed. The skip-gram model is built by flipping the continuous bag-of-words model. Instead of trying to predict the middle word given $2n$ surrounding words, the skip-gram model tries to predict randomly chosen one of the $2n$ surrounding words given the middle word. From this description alone, it is quite clear that this skip-gram model is not going to be great as a language model. However, it turned out that the word vectors obtained by training a skip-gram model were as good as those obtained by either a continuous bag-of-words model or any other neural language model. Of course, it is debatable which criterion be used to determine the goodness of word vectors, but in many of the existing so-called "intrinsic" evaluations, those obtained from a skip-gram model have been shown to excel.

The authors of [62] recently showed that training a skip-gram model with negative sampling (see [68]) is equivalent to factorizing a positive point-wise mutual information matrix (PPMI) into two lower-dimensional matrices. The left lower-dimensional matrix corresponds to the input word embedding matrix $\mathbf{E}$ in a skip-gram model. In other words, training a skip-gram model *implicitly* factorizes a PPMI matrix.

Their work drew a nice connection between the existing works on distributional word representations from natural language processing, or even computational linguistics and these more recent neural approaches. I will not go into any further detail in this course, but I encourage readers to read [62].

5.4.3 Semi-Supervised Learning with Pretrained Word Embeddings

One thing I want to emphasize in these language models, including n-gram language model, neural language model and continuous bag-of-words model, is that they are purely *unsupervised*, meaning that all we need is a large corpus of unannotated text. This is one thing that makes this statistical approach to language modelling much more appealing than any other approach based on linguistic structures (see Sec. 5.1.1 for a brief discussion.)

[20] One difference between the model we derived in this section starting from the MRF-LM and the one proposed in [68] is that in our derivation, the neural network shares a single weight matrix $\mathbf{E}$ for both the input and output layers.

When it comes to neural language model and continuous bag-of-words model, we now know that these networks learn continuous vector representations of input words, target words and the context phrase ($\mathbf{h}$ from Eq. (5.12).) We also discussed how these vector representations encode similarities among different linguistic units, be it a word or a phrase.

What this implies is that once we train this type of language model on a large, or effectively infinite,[21] corpus of *unlabelled* text, we get good vectors for those linguistic units for free. Among these, word vectors, the rows of the input weight matrix $\mathbf{E}$ in Eq. (5.11), have been extensively used in many natural language processing applications in recent years since [88, 30, 68].

Let us consider an extreme example of classifying each English word as either "positive" or "negative". For instance, "happy" is positive, while "sad" is negative. A training set of 2 examples–1 positive and 1 negative words– is given. How would one build a classifier?[22]

There are two issues here. First, it is unclear how we should represent the input, in this case a word. A good reader who has read this note so far will be clearly ready to use an one-hot vector and use a softmax layer in the output, and I commend you for that. However, this still does not solve a more serious issue which is that we have only *two* training examples! All the word vectors, save for two vectors corresponding to the words in the training set, will not be updated at all.

One way to overcome these two issues is to make somewhat strong, but reasonable assumption that *similar* input will have similar sentiments. This assumption is at the heart of semi-supervised learning [21]. It says that high-dimensional data points in effect lies on a lower-dimensional manifold, and the target values of the points on this manifold change smoothly. Under this assumption, if we can well model this lower-dimensional data manifold using unlabelled training examples, we can train a good classifier[23]

And, guess what? We have access to this lower-dimensional manifold, which is represented by the set of *pretrained* word vectors $\mathbf{E}$. Believing that similar words have similar sentiment and that these pretrained word vectors indeed well reflect similarities among words, let me build a simple nearest neighbour (NN) classifier which uses the pretrained word vectors:

$$\mathrm{NN}(w) = \begin{cases} \text{positive}, & \text{if } \cos(\mathbf{e}_w, \mathbf{e}_{\text{happy}}) > \cos(\mathbf{e}_w, \mathbf{e}_{\text{bad}}) \\ \text{negative}, & \text{otherwise} \end{cases},$$

where $\cos(\cdot,\cdot)$ is a cosine similarity defined as

$$\cos(\mathbf{e}_i, \mathbf{e}_j) = \frac{\mathbf{e}_i^\top \mathbf{e}_j}{\|\mathbf{e}_i\| \|\mathbf{e}_j\|}.$$

[21] Why? Because of almost universal broadband access to the Internet!

[22] Although the setting of 2 training examples is extreme, but the task itself turned out to be not-so-extreme. In fact, there is multiple dictionaries of words' sentiment maintained. For instance, check `http://sentiwordnet.isti.cnr.it/search.php?q=llama`.

[23] What do I mean by a good classifier? A good classifier is a classifier that classifies *unseen* test examples well. See Sec. 2.3.

This use of a term "similarity" almost makes this set of pretrained word vectors look like some kind of magical wand that can solve everything.[24] This is however not true, and using pretrained word vectors must be done with caution.

Why should we be careful in using these pretrained word vectors? We must remember that these word vectors were obtained by training a neural network to maximize a certain objective, or to minimize a certain cost function. This means that these word vectors capture certain aspects of words' underlying structures that are necessary to achieve the training objective, and that there is no reason for these word vectors to capture any other properties of the words that are not necessary for maximizing the training objective. In other words, "similarity" among multiple words has many different aspects, and these word vectors will capture only a few of these many aspects. Which few aspects will be determined by the choice of training objective.

The hope is that language modelling is a good training objective that will encourage the word vectors to capture as many aspects of similarity as possible.[25] But, is this true in general?

Let's consider an example of words describing emotions, such as "happy", "sad" and "angry", in the context of a continuous bag-of-words model. These emotion-describing words often follow some forms of a verb "feel", such as "feel", "feels", "felt" and "feeling". This means that those emotion-describing words will have to be projected nearby in the context space in order to give a high probability to those forms of "feel" as a middle word. This is understandable and agrees quite well with our intuition. All those emotion-describing words are similar to each other in the sense that they all describe emotion. But, wait, this aspect of similarity is not going to help sentiment classification of words. In fact, this aspect of similarity will *hurt* the sentiment classifier, because a positive word "happy" will be close to negative words "sad" and "angry" in this word vector space!

The lesson here is that when you are solving a language-related task with very little data, it is a good idea to consider using a set of pretrained word vectors from neural language models. However, you must do so in caution, and perhaps try to pretrain your own word vectors by training a neural network to maximize a certain objective that better suits your final task.

But, then, what other training objectives are there? We will get to that later.

5.5 Recurrent Language Model

Neural language model indeed avoids the lack of generalization in the conventional n-gram language modeling. It still assumes the n-th order Markov property, meaning that it looks only as far back into the past as $n-1$ words. In Sec. 5.3, I gave an example of "In Korea, more than half of all the residents speak Korean". In this example, the conditional distribution over the last word in the sentence clearly will be better estimated

[24] For future reference, I must say there were many papers claiming that the pretrained word vectors are indeed magic wands at three top-tier natural language processing conferences (ACL, EMNLP, NAACL) in 2014 and 2015.

[25] Some may ask how a single vector, which is a point in a space, can capture multiple aspects of similarity. This is possible because these word vectors are high-dimensional.

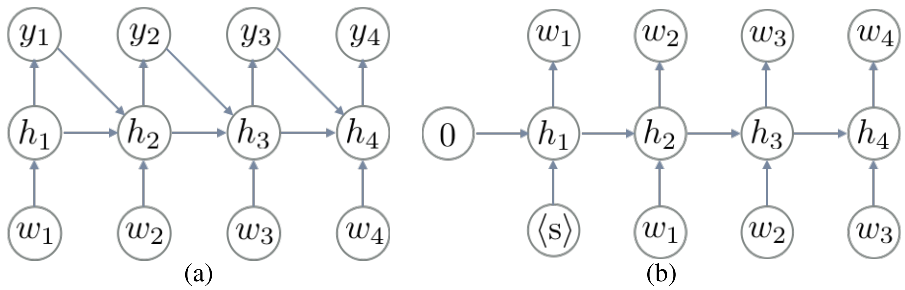


Figure 5.7: (a) A recurrent neural network from Sec. 4.1.4. (b) A recurrent neural network language model.

if it is conditioned on the second word of the sentence which is more than 10 words back in the past.

Let us recall what we learned in Sec. 4.1.4. There, we learn how to build a recurrent neural network to read a variable-length sequence and return a variable-length output sequence. An example we considered back then was a task of part-of-speech tagging, where the input is a sentence such as

$$x = (\text{Children}, \text{eat}, \text{sweet}, \text{candy}),$$

and the target output is a sequence of part-of-speech tags such as

$$y = (\text{noun}, \text{verb}, \text{adjective}, \text{noun}).$$

In order to make less of an assumption on the conditional independence of the predicted tags, we made a small adjustment such that the prediction Y_t at each timestep was fed back into the recurrent neural network in the next timestep together with the input X_{t+1}. See Fig. 5.7 (a) for graphical illustration.

Why am I talking about this again, after saying that the task of part-of-speech tagging is not even going to be considered as a valid topic for the final project? Because the very same model for part-of-speech tagging will be turned into the very *recurrent neural network language model* in this section.

Let us start by considering a single conditional distribution, marked (a) below, from the full sentence probability:

$$p(w^1, w^2, \ldots, w^T) = \prod_{t=1}^{T} \underbrace{p(w^t | w^1, \ldots, w^{t-1})}_{(a)}.$$

This conditional probability can be approximated by a neural network, as we've been doing over and over again throughout this course, that takes as input $(w^1, \ldots, w^{t-1})$ and returns the probability over all possible words in the vocabulary V. This is not unlike neural language model we discussed earlier in Sec. 5.4, except that the input is now a variable-length sequence.

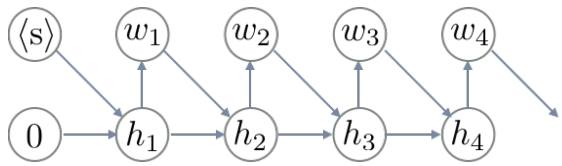


Figure 5.8: A recurrent neural network language model

In this case, we can use a recurrent neural network which is capable of summarizing/memorizing a variable-length input sequence. A recurrent neural network summarizes a given input sequence $(w^1, \ldots, w^{t-1})$ into a memory state $\mathbf{h}^{t-1}$:

$$\mathbf{h}^{t'} = \begin{cases} 0, & \text{if } t' = 0 \\ f(\mathbf{e}_{w^{t'}}, \mathbf{h}^{t'-1}), & \text{otherwise} \end{cases}, \tag{5.16}$$

where t' runs from 0 to $t-1$. f is a recurrent function which can be any of a naive transition function from Eq. (4.1), a gated recurrent unit or a long short-term memory unit from Sec. 4.2.2. $\mathbf{e}_{w^{t'}}$ is a word vector corresponding to the word $w^{t'}$.

This summary $\mathbf{h}^{t-1}$ is affine-transformed followed by a softmax nonlinear function to compute the conditional probability of w^t. Hopefully, everyone remembers how it is done. As in Eq. (4.5),

$$\mu = \text{softmax}(\mathbf{V}\mathbf{h}^{t-1}),$$

where μ is a vector of probabilities of all the words in the vocabulary.

One thing to notice here is that the iteration procedure in Eq. (5.16) computes a sequence of every memory state vector $\mathbf{h}^t$ by simply reading the input sentence *once*. In other words, we can let the recurrent neural network read one word w^t at a time, update the memory state $\mathbf{h}^t$ and compute the conditional probability of the *next* word $p(w^{t+1}|w^{\leq t})$.

This procedure is illustrated in Fig. 5.7 (b).[26] This language model is called a *recurrent neural network language model* (RNN-LM, [69]).

But, wait, from looking at Figs. 5.7 (a)–(b), there is a clear difference between the recurrent neural networks for part-of-speech tagging and language model. That is, there is no feedback connection from the output of the previous time step back into the recurrent neural network in the RNN-LM. This is simply an illusion from the limitation in the graphical illustration, because the input w^{t+1} in the next time step is in fact the output w^{t+1} at the current time step. This becomes clearer by drawing the same figure in a slightly different way, as in Fig. 5.8.

[26] In the figure, you should notice the beginning-of-the-sentence symbol ⟨s⟩. This is necessary in order to use the very same recurrent function f to compute the conditional probability of the first word in the input sentence.

5.6 How do n-gram language model, neural language model and RNN-LM compare?

Now the question is which one of these language models we should use in practice. In order to answer this, we must first discuss the metric most commonly used for evaluating language models.

The most commonly used metric is a *perplexity*. In the context of language modelling, the perplexity PPL of a model $\mathscr{M}$ is computed by

$$\text{PPL} = b^{-\frac{1}{N}\sum_{n=1}^{N} \log_b p_{\mathscr{M}}(w_n|w_{<n})}, \tag{5.17}$$

where N is the number of all the words in the validation/test corpus, and b is some constant that is often 2 or 10 in practice.

What is this perplexed metric? I totally agree with you on this one. Of course, there is a quite well principled way to explain what this perplexity is based on information theory. This is however not necessary for us to understand this metric called perplexity.

As the exponential function (with base b in the case of perplexity in Eq. (5.17)) is a monotonically increasing function, we see that the ordering of different language models based on the perplexity will not change even if we only consider the exponent:

$$-\frac{1}{N}\sum_{n=1}^{N} \log_b p_{\mathscr{M}}(w_n|w_{<n}).$$

Furthermore, assuming that $b > 1$, we can simply replace $\log_b$ with log (natural logarithm) without changing the order of different language models:

$$-\frac{1}{N}\sum_{n=1}^{N} \log p_{\mathscr{M}}(w_n|w_{<n}).$$

Now, this looks awfully similar to the cost function, or negative log-likelihood, we minimize in order to train a neural network (see Chapter 2.)

Let's take a look at a single term inside the summation above:

$$\log p_{\mathscr{M}}(w_n|w_{<n}).$$

This is simply measuring how high a probability the language model $\mathscr{M}$ is assigning to a *correct next word* given all the previous words. Again, because log is a monotonically increasing function.

In summary, the (inverse) perplexity measures how high a probability the language model $\mathscr{M}$ assigns to correct next words in the test/validation corpus *on average*. Therefore, *a better language model is the one with a lower perplexity*. There is nothing so perplexing about the perplexity, once we start viewing it from this perspective.

We are now ready to compare different language models, or to be more precise, three different classes of language models–count-based n-gram language model, neural n-gram language model and recurrent neural network language model. The biggest challenge in doing so is that this comparison will depend on many factors that are not easy to control. To list a few of them,

- Language
- Genre/Topic of training, validation and test corpora
- Size of a training corpus
- Size of a language model

LM	Hidden Layers	PPL	CER	WER
Count-based	–	131.2	7.6	12.4
+ Feedforward	100	121.1	7.5	11.8
	200	116.6	7.3	11.6
	300	114.7	7.3	11.5
	400	114.1	7.2	11.5
	500	113.4	7.2	11.5
	600	112.5	7.2	11.5
	2x 100	121.2	7.5	11.9
	2x 200	115.7	7.3	11.5
	2x 300	115.4	7.2	11.5
	2x 400	112.2	7.1	11.3
	2x 500	111.0	7.1	11.3
	2x 600	110.2	7.2	11.3
+ RNN	100	121.0	7.5	11.8
	200	117.6	7.3	11.7
	300	112.6	7.3	11.4
	400	111.5	7.2	11.3
	500	108.9	7.1	11.2
	600	108.1	7.0	11.1
+ LSTM	100	115.3	7.3	11.7
	200	106.8	7.1	11.2
	300	102.4	6.9	11.0
	400	99.9	6.9	10.9
	500	97.9	6.9	10.9
	600	96.7	6.8	10.8
	2x 100	111.0	7.2	11.4
	2x 200	101.6	7.0	11.0
	2x 300	97.5	6.8	10.8
	2x 400	95.2	6.9	10.8
	2x 500	93.1	**6.6**	10.5
	2x 600	**92.0**	6.7	**10.4**

Figure 5.9: The perplexity, word error rate (WER) and character error rate (CER) of an automatic speech recognition system using different language models. Note that all the results by neural or recurrent language models are by interpolating these models with the count-based n-gram language model. Reprinted from [86].

Because of this difficulty, this kind of comparison has often been done in the context of a specific downstream application. This choice of a downstream application often puts rough constraints on the size of available, or commonly used, corpus, target language and reasonably accepted size of language models. For instance, the authors of [2] compared the conventional n-gram language model and neural language model, with various approximation techniques, with machine translation as a final task. In [86], the authors compared all the three classes of language model in the context of automatic speech recognition.

First, let us look at one observation made in [86]. From Fig. 5.9, we can see that it is beneficial to use a recurrent neural network language model (RNN-LM) compared

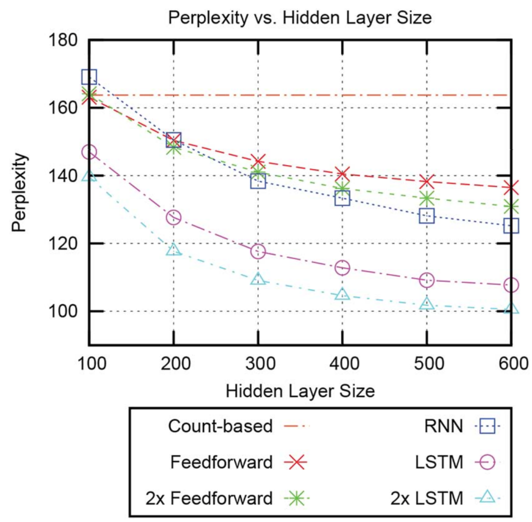


Figure 5.10: The trend of perplexity as the size of language model changes. Reprinted from [86].

to a usual neural language model. Especially when long short-term memory units were used, the improvement over the neural language model was significant. Furthermore, we see that it is possible to improve these language models by simply increasing their size.

Similarly, in Fig. 5.10 from the same paper [86], it is observed that larger language models tend to get better/lower perplexity and that RNN-LM in general outperforms neural language models.

These two observations do seem to suggest that neural and recurrent language models are better candidates as language model. However, this is not to be taken as an evidence for choosing neural or recurrent language models. It has been numerously observed over years that the best performance, both in terms of perplexity and in terms of performance in the downstream applications such as machine translation and automatic speech recognition, is achieved by combining a count-based n-gram language model and a neural, or recurrent, language model. See, for instance, [81].

This superiority of combined, or hybrid, language model suggests that the count-based, or conventional, n-gram language model, neural language model and recurrent neural network language model are capturing underlying structures of natural language sentences that are complement to each other. However, it is not crystal clear how these captured structures differ from each other.

Chapter 6

Neural Machine Translation

Finally, we have come to the point in this course where we discuss an *actual* natural language task. In this chapter, we will discuss how translation from one language to another can be done with statistical methods, more specifically neural networks.

6.1 Statistical Approach to Machine Translation

Let's first think of what it means to translate one sentence X in a source language to an equivalent sentence Y in a target language which is different from the source language. A process of translation is a function that takes as input the source sentence X and returns a correct translation Y, and it is clear that there may be more than one correct translations. The latter fact implies that this function of translation should return not a single correct translation, but a probability distribution that assigns high probabilities to more than one likely translations.

Now, let us write it in a more formal way. First, the input is a sequence of words

$$X = (x_1, x_2, \ldots, x_{T_x}),$$

where T_x is the length of the source sentence. A target sentence is

$$Y = (y_1, y_2, \ldots, y_{T_y}).$$

Similarly, T_y is the length of the target sentence.

The translation function f then reads the input sequence X and computes the probability over target sentences. In other words,

$$f : V_x^+ \to C^+_{|V_y|-1} \tag{6.1}$$

where V_x is a source vocabulary, and V_x^+ is a set of all possible source sentences of any length $T_x > 0$. V_y is a target vocabulary, and C_k is a standard k-simplex.

What is a standard k-simplex? It is a set defined by

$$C_k = \left\{ (t_0, \ldots, t_k) \in \mathbb{R}^{k+1} \left| \sum_{i=1}^{k} t_k = 1 \text{ and } t_i \geq 0 \text{ for all } i \right. \right\}.$$

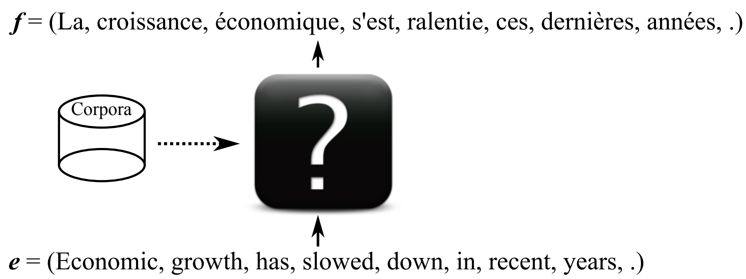


Figure 6.1: Graphical illustration of statistical machine translation

In short, this set contains all possible settings for categorical distributions of $k+1$ possible outcomes. This means that the translation function f returns a probability distribution $P(Y|X)$ over all possible translations of length $T_y > 1$.

Given a source sentence X, this translation function f returns the conditional probability of a translation Y: $P(Y|X)$. Let us rewrite this conditional probability according to what we have discussed in Chapter 5:

$$P(Y|X) = \prod_{t=1}^{T_y} \underbrace{P(y_t|y_1,\ldots,y_{t-1}, \underbrace{X}_{\text{conditional}})}_{\text{language modelling}} \tag{6.2}$$

Looking at it in this way, it is clear that this is nothing but *conditional language modelling*. This means that we can use any of the techniques we have used earlier in Chapter 5 for statistical machine translation.

Training can be trivially done by maximizing the log-likelihood or equivalently minimizing the *negative* log-likelihood (see Sec. 3.1):

$$\tilde{C}(\theta) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_y} \log p(y_t^n|y_{<t}^n, X^n), \tag{6.3}$$

given a training set

$$D = \left\{(X^1,Y^1),(X^2,Y^2),\ldots,(X^N,Y^N)\right\} \tag{6.4}$$

consisting of N training *pairs*.

All these look extremely straightforward and do not deviate too much from what we have learned so far in this course. A big picture on this process translation is shown in Fig. 6.1. More specifically, building a statistical machine translation model is simple, because we have learned how to

1. Assign a probability to a sentence in Sec. 5.2.
2. Handle variable-length sequences with recurrent neural networks in Sec. 4.1.
3. Compute the gradient of an empirical cost function $\tilde{C}$ with respect to the parameters θ of a recurrent neural network in Sec. 4.1.2 and Sec. 3.4.

4. Use stochastic gradient descent to minimize the cost function in Sec. 2.2.2.

Of course, simply knowing all these does not get you a working neural network that translates from one language to another. We will discuss in detail how we can build such a neural network in the next section. Before going to the next section, we must first discuss two issues; (1) where do we get training data? (2) how do we evaluate machine translation systems?

6.1.1 Parallel Corpora: Training Data for Machine Translation

First, let us consider again what the problem we're trying to solve here. It is machine translation, and from the description in the previous section and from Eqs. (6.1)–(6.2), it is a *sentence-to-sentence* translation task. We approach this problem by building a model that takes as input a source sentence S and computes the probability $P(Y|X)$ of a target sentence Y, equivalently a translation. In order for this model to translate, we must train it with a training set of pairs of a source sentence and its correct translation.

The very first problem we run into is where we can find this training set which is often called a *parallel corpus*. It is not easy to think of documents which have been translated into multiple languages. Let's take for instance all the books that are being translated each year. According to [75], approximately 3% of titles published each year in English are translations from another language.[1] A few international news agencies publish some of their news articles in multiple languages. For instance, AFP publishes 1,500 stories in French, 700 stories in English, 400 stories in Spanish, 250 stories in Arabic, 200 stories in German and 150 stories in Portuguese each day, and there are some overlapping stories across these six languages.[2] Online commerce sites, such as eBay, often list their products in international sites with their descriptions in multiple languages.[3]

Unfortunately these sources of multiple languages of the same content are not suitable for our purpose. Why is this so? Most importantly, they are often copy-righted and sold for personal use only. We cannot buy more than 14,400 books in order to train a translation model. We will likely go broke before completing the purchase, and even if so, it is unclear whether it is acceptable under copyright to use these text to build a translation model. Because we are mixing multiple sources of which each is protected under copyright, is the translation model trained from a mix of all these materials considered a derivative work?[4]

This issue is nothing new, and has been there since the very first statistical machine translation system was proposed in [17]. Fortunately, it turned out that there are a number of legitimate sources where we can get documents translated in more than one languages, often very faithfully to their content. These sources are parliamentary proceedings of bilingual, or multilingual countries.

[1] "According to the information Bowker released in October of 2005, in 2004 there were 375,000 new books published in English." .. "Of that total, approx. 14,440 were new translations, which is slightly more than 3% of all books published." [75].

[2] `http://www.afp.com/en/products/services/text`

[3] `http://sellercentre.ebay.co.uk/international-selling-tools`

[4] `http://copyright.gov/circs/circ14.pdf`

Brown et al. [17] used the proceedings from the Canadian parliament, which are by law kept in both French and English. All of these proceedings are digitally available and called *Hansards*. You can check it yourself online at http://www.parl.gc.ca/, and here's an excerpt from the Prayers of the 2nd Session, 41st Parliament, Issue 152:[5]

- **French**: "ELIZABETH DEUX, par la Grâce de Dieu, REINE du Royaume-Uni, du Canada et de ses autres royaumes et territoires, Chef du Commonwealth, Défenseur de la Foi."
- **English**: "ELIZABETH THE SECOND, by the Grace of God of the United Kingdom, Canada and Her other Realms and Territories QUEEN, Head of the Commonwealth, Defender of the Faith."

Every single word spoken in the Canadian parliament is translated either into French or into English. A more recent version of Hansards preprocessed for research can be found at http://www.isi.edu/natural-language/download/hansard/.

Similarly, the European parliament used to provided the parliamentary proceedings in all 23 official languages.[6] This is a unique data in the sense that each and every sentence is translated into either 11 or 26 official languages. For instance, here is one example [55]:

- **Danish**: det er næsten en personlig rekord for mig dette efterår.
- **German**: das ist für mich fast persönlicher rekord in diesem herbst .
- **Greek**: (omitted)
- **English**: that is almost a personal record for me this autumn !
- **Spanish**: es la mejor marca que he alcanzado este otoño .
- **Finnish**: se on melkein minun ennätykseni tänä syksynä !
- **French**: c ' est pratiquement un record personnel pour moi , cet automne !
- **Italian**: e ' quasi il mio record personale dell ' autunno .
- **Dutch**: dit is haast een persoonlijk record deze herfst .
- **Portuguese**: é quase o meu recorde pessoal deste semestre !
- **Swedish**: det är nästan personligt rekord för mig denna höst !

The European proceedings has been an invaluable resource for machine translation research. At least, the existing multilingual proceedings (up to 2011) can be still used, and it is known in the field as the "Europarl" corpus [55] and can be downloaded from http://www.statmt.org/europarl/.

These proceedings-based parallel corpora have two distinct advantages. First, in many cases, the sentences in those corpora are well-formed, and their translations are

[5] This is one political lesson here: *Canada is still headed by the Queen of the United Kingdom*.

[6] Unfortunately, the European parliament decided to stop translating its proceedings into all 23 official languages on 21 Nov 2011 as an effort toward budget cut. See http://www.euractiv.com/culture/parliament-cuts-translation-budg-news-516201.

done by professionals, meaning the quality of the corpora is guaranteed. Second, surprisingly, the topics discussed in those proceedings are quite diverse. Clearly the members of the parliament do not often chitchat too often, but they do discuss a diverse set of topics. Here's one such example from the Europarl corpus:

- **English**: Although there are now two Finnish channels and one Portuguese one, there is still no Dutch channel, which is what I had requested because Dutch people here like to be able to follow the news too when we are sent to this place of exile every month.
- **French**: Il y a bien deux chaînes finnoises et une chaîne portugaise, mais il n'y a toujours aucune chaîne néerlandaise. Pourtant je vous avais demandé une chaîne néerlandaise, car les Néerlandais aussi désirent pouvoir suivre les actualités chaque mois lorsqu'ils sont envoyés en cette terre d'exil.

One apparent limitation is that these proceedings cover only a handful of languages in the world, mostly west European languages. This is not desirable. Why? According to Ethnologue (2014)[7], the top-five most spoken languages in the world are

1. Chinese: approx. 1.2 billion
2. Spanish: approx. 414 million
3. English: approx. 335 million
4. Hindi: approx. 260 million
5. Arabic: approx. 237 million

There are only two European languages in this list.

So, then, where can we get all data for all these non-European languages? There are a number of resources you can use, and let me list a few of them here:

You can find the translated subtitle of the TED talks at the Web Inventory of Transcribed and Translated Talks (WIT^3, `https://wit3.fbk.eu/`) [20]. It is a quite small corpus, but includes 104 languages. For Russian–English data, Yandex released a parallel corpus of one million sentence pairs. You can get it at `https://translate.yandex.ru/corpus?lang=en`. You can continue with other languages by googling very hard, but eventually you run into a hard wall.

This hard wall is not only the lack of any resource, but also lack of enough resource. For instance, I quickly googled for Korean–English parallel corpora and found the following resources:

- SWRC English-Korean multilingual corpus: 60,000 sentence pairs `http://semanticweb.kaist.ac.kr/home/index.php/Corpus10`
- Jungyeul's English-Korean parallel corpus: 94,123 sentence pairs `https://github.com/jungyeul/korean-parallel-corpora`

This is just not large enough.

One way to avoid this or mitigate this problem is to automatically mine parallel corpora from the Internet. There have been quite some work in this direction as a way

[7] `http://www.ethnologue.com/world`

to increase the size of parallel corpora [76, 93]. The idea is to build an algorithm that crawls the Internet and find a pair of corresponding pages in two different languages. One of the largest preprocessed corpus of multiple languages from the Internet is the Common Crawl Parallel Corpus created by Smith et al. [84] available at `http://www.statmt.org/wmt13/training-parallel-commoncrawl.tgz`.

6.1.2 Automatic Evaluation Metric

Let's say we have trained a machine translation model on a training corpus. A big question follows: *how do we evaluate this model*?

In the case of classification, evaluation is quite straightforward. All we need to do is to classify held-out test examples with a trained classifier and see how many examples were correctly classified. This is however not true in the case of translation.

There are a number of issues, but let us discuss two most important problems here. First, there may be many correct translations given a single source sentence. For instance, the following three sentences are the translations made by a human translator given a single Chinese sentence [71]:

- It is a guide to action that ensures that the military will forever heed Party commands.
- It is the guiding principle which guarantees the military forces always being under the command of the Party.
- It is the practical guide for the army always to heed the directions of the party.

They all clearly differ from each other, although they are the translations of a single source sentence.

Second, the quality of translation cannot be measured as either success or failure. It is rather a smooth measure between success and failure. Let us consider an English translation of a French sentence "J'aime un llama, qui est un animal mignon qui vit en Amérique du Sud".[8]

One possible English translation of this French sentence is "I like a llama which is a cute animal living in South America". Let's give this translation a score 100 (success). According to Google translate, the French sentence above is "I like a llama, a cute animal that lives in South America". I see that Google translate has omitted "qui est" from the original sentence, but the whole meaning has well been captured. Let us give this translation a slightly lower score of 90.

Then, how about "I like a llama from South America"? This is certainly not a correct translation, but except for the part about a llama being cute, this sentence does communicate most of what the original French sentence tried to communicate. Maybe, we can give this translation a score of 50.

How about "I do not like a llama which is an animal from South America"? This translation correctly describes the characteristics of llama exactly as described in the source sentence. However this translation incorrectly states that I do not like a llama, when I like a llama according to the original French sentence. What kind of score would you give this translation?

[8] I would like to thank Laurent Dinh for the French translation.

Even worse, we want an automated evaluation algorithm. We cannot look at thousands of validation or test sentence pairs to tell how well a machine translation model does. Even if we somehow did it for a single model, in order to compare this translation model against others, we must do it for every single machine translation model under comparison. We must have an automatic evaluation metric in order to efficiently test and compare different machine translation models.

BLEU One of the most widely used automatic evaluation metric for assessing the quality of translations is BLEU proposed in [71]. BLEU computes the geometric mean of the modified n-gram precision scores multiplied by brevity penalty. Let me describe this in detail here.

First, we define the modified n-gram precision p_n of a translation Y as

$$p_n = \frac{\sum_{S \in C} \sum_{\text{ngram} \in S} \hat{c}(\text{ngram})}{\sum_{S \in C} \sum_{\text{ngram} \in S} c(\text{ngram})},$$

where C is a corpus of all the sentences/translations, and S is a set of all unique n-grams in one sentence in C. $c(\text{ngram})$ is the count of the n-gram, and $\hat{c}(\text{ngram})$ is

$$\hat{c}(\text{ngram}) = \min(c(\text{ngram}), c_{\text{ref}}(\text{ngram})).$$

$c_{\text{ref}}(\text{ngram})$ is the count of the n-gram in reference sentences.

What does this modified n-gram precision measure? It measures the ratio between the number of n-grams in the translation and the number of those n-grams actually occurred in a reference (ground-truth) translation. If there is no n-gram from the translation in the reference, this modified precision will be zero because $c_{\text{ref}}(\cdot)$ will be zero all the time.

It is common to use the geometric average of modified 1-, 2-, 3- and 4-gram precisions, which is computed by

$$P_1^4 = \exp\left(\frac{1}{4} \sum_{n=1}^{4} \log p_n\right).$$

If we use this geometric average P as it is, there is a big loophole. One can get a high average modified precision by making as short a translation as possible. For instance, a reference translation is

- I like a llama, a cute animal that lives in South America .

and a translation we are trying to evaluate is

- cute animal that lives

This is clearly a very bad translation, but the modified 1-, 2-, 3- and 4-gram precisions will be high. The modified precisions are

$$p_1 = \frac{1+1+1+1}{1+1+1+1} = \frac{4}{4} = 1$$
$$p_2 = \frac{1+1+1}{1+1+1} = \frac{3}{3} = 1$$
$$p_3 = \frac{1+1}{1+1} = \frac{2}{2} = 1$$
$$p_4 = \frac{1}{1} = \frac{1}{1} = 1.$$

Their geometric average is then

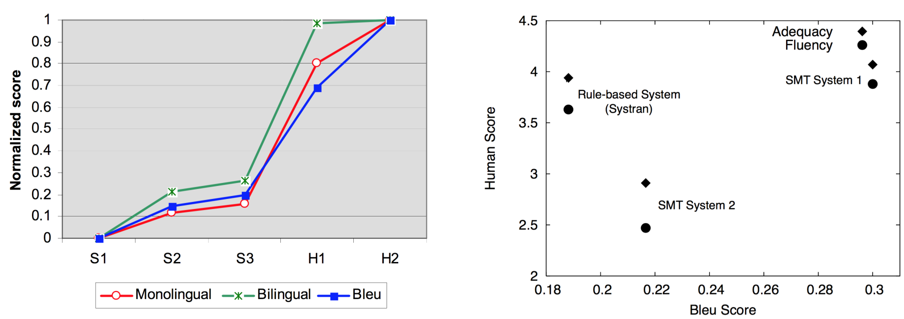
$$P_1^4 = \exp\left(\frac{1}{4}(0+0+0+0)\right) = 1$$

which is the maximum modified precision you can get!

In order to avoid this behaviour, BLEU penalizes the geometric average of the modified n-gram precisions by the ratio of the lengths between the reference r and translation l. This is done by first computing a brevity penalty:

$$\mathrm{BP} = \begin{cases} 1 & \text{, if } l \geq r \\ \exp\left(1 - \frac{r}{l}\right) & \text{, if } l < r \end{cases}$$

If the translation is longer than the reference, it uses the geometric average of the modified n-gram precisions as it is. Otherwise, it will penalize it by multiplying the average precision with a scalar less than 1. In the case of the example above, the brevity penalty is 0.064, and the final BLEU score is 0.064.



(a) (b)

Figure 6.2: (a) BLEU vs. bilingual and monolingual judgements of three machine translation systems (S1, S2 and S3) and two humans (H1 and H2). Reprinted from [71]. (b) BLEU vs. human judgement (adequacy and fluency separately) of three machine translation systems (two statistical and one rule-based systems). Reprinted from [18].

The BLEU was shown to correlate well with human judgements in the original article [71]. Fig. 6.2 (a) shows how BLEU correlates with the human judgements in comparing different translation systems.

This is however not to be taken as a message saying that the BLEU is the perfect automatic evaluation metric. It has been shown that the BLEU is only adequate in comparing two similar machine translation systems, but not too much so in comparing two very different systems. For instance, Callison-Burch et al. [18] observed that the BLEU underestimates the quality of the machine translation system that is *not* a phrase-based statistical system. See Fig. 6.2 (b) for an example.

BLEU is definitely not a perfect metric, and many researchers strive to build a better evaluation metric for machine translation systems. Some of the alternatives available at the moment are METEOR [33] and TER [85].

6.2 Neural Machine Translation: Simple Encoder-Decoder Model

From the previous section and from Eq. 6.2, it is clear that we need to model each conditional distribution inside the product as a function. This function will take as input all the previous words in the target sentence $Y = (y_1, \ldots, y_{t-1})$ and the whole source sentence $X = (x_1, \ldots, x_{T_x})$. Given these inputs the function will compute the probabilities of all the words in the target vocabulary V_y. In this section, I will describe an approach that was proposed multiple times independently over 17 years in [38, 25, 87].

Let us start by tackling how to handle the source sentence $X = (x_1, \ldots, x_{T_x})$. Since this is a variable-length sequence, we can readily use a recurrent neural network from Chapter 4. However, unlike the previous examples, there is no explicit target/output in this case. All we need is a (vector) summary of the source sentence.

We call this recurrent neural network an *encoder*, as it encodes the source sentence into a (continuous vector) code. It is implemented as

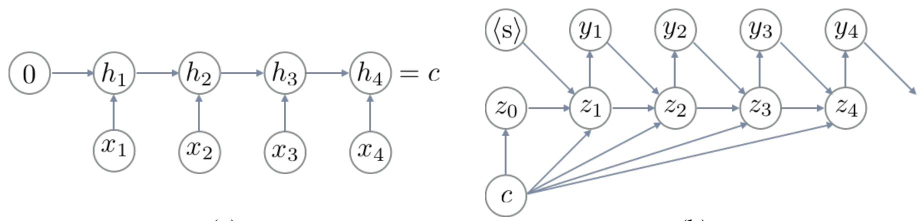
$$\mathbf{h}_t = \phi_{\text{enc}} \left(\mathbf{h}_{t-1}, \mathbf{E}_x^\top \mathbf{x}_t \right). \tag{6.5}$$

As usual, ϕ_{enc} can be any recurrent activation function, but it is highly recommended to use either gated recurrent units (see Sec. 4.2.2) or long short-term memory units (see Sec. 4.2.3.) $\mathbf{E}_x \in \mathbb{R}^{|V_x| \times d}$ is an input weight matrix containing word vectors as its rows (see Eq. (5.11) in Sec. 5.4,) and $\mathbf{x}_t$ is an one-hot vector representation of the word x_t (see Eq. (5.10) in Sec. 5.4.) $\mathbf{h}_0$ is initialized as an all-zero vector.

After reading the whole sentence up to $\mathbf{x}_{T_x}$, the last memory state $\mathbf{h}_{T_x}$ of the encoder summarizes the whole source sentence into a single vector, as shown in Fig. **??** (a). Thanks to this encoder, we can now work with a single vector instead of a whole sequence of source words. Let us denote this vector as $\mathbf{c}$ and call it a *context vector*.

We now need to design a *decoder*, again, using a recurrent neural network. As I mentioned earlier, the decoder is really nothing but a language model, except that it is conditioned on the source sentence X. What this means is that we can build a recurrent neural network language model from Sec. 5.5 but feeding also the context vector at each time step. In other words,

$$\mathbf{z}_t = \phi_{\text{dec}} \left(\mathbf{z}_{t-1}, \left[\mathbf{E}_y^\top \mathbf{y}_{t-1}; \mathbf{c} \right] \right) \tag{6.6}$$



(a) (b)

Figure 6.3: (a) The encoder and (b) the decoder of a simple neural machine translation model

Do you see the similarity and dissimilarity to Eq. (5.16) from Sec. 5.5? It's essentially same, except that the input at time t is a concatenated vector of the word vector of the previous word y_{t-1} and the context vector $\mathbf{c}$.

Once the decoder's memory state is updated, we can compute the probabilities of all possible target words by

$$p(y_t = w' | y_{<t}, X) \propto \exp\left(\mathbf{e}_{w'}^{\top} \mathbf{z}_t\right), \tag{6.7}$$

where $\mathbf{e}_{w'}$ is the target word vector associated the word w'. This is equivalent to affine-transforming $\mathbf{z}_t$ followed by a softmax function from Eq. (3.5) from Sec. 3.1.

Now, should we again initialize $\mathbf{z}_0$ to be an all-zero vector? Maybe, or maybe not. One way to view what this decoder does is that the decoder models a *trajectory* in a continuous vector space, and each point in the trajectory is $\mathbf{z}_t$. Then, $\mathbf{z}_0$ acts as a starting point of this trajectory, and it is natural to initialize this starting point to be a point relevant to the source sentence. Because we have access to the source sentence's content via $\mathbf{c}$, we can again use it to initialize $\mathbf{z}_0$ as

$$\mathbf{z}_0 = \phi_{\text{init}}(\mathbf{c}). \tag{6.8}$$

See Fig. 6.3 (b) for the graphical illustration of the decoder.

Although I have used $\mathbf{c}$ as if it is a separate variable, this is not true. $\mathbf{c}$ is simply a shorthand notation of the last memory state of the encoder which is a function of the whole source sentence. What does this mean? It means that we can compute the gradient of the empirical cost function in Eq. (6.3) with respect to all the parameters of both the encoder and decoder and maximize the cost function using stochastic gradient descent, just like any other neural network we have learned so far in this course.

6.2.1 Sampling vs. Decoding

Sampling We are ready to compute the conditional distribution $P(Y|X)$ over all possible translations given a source sentence. When we have a distribution, the first thing we can try is to sample from this distribution. Often, it is not straightforward to generate samples from a distribution, but fortunately, in this case, we can readily generate exact samples from the distribution $P(Y|X)$.

We simply iterate over the following steps until a token indicating the end of a sentence ($\langle\text{eos}\rangle$):

1. Compute $\mathbf{c}$ (Eq. (6.5))
2. Initialize $\mathbf{z}_0$ with $\mathbf{c}$ (Eq. (6.8))
3. Compute $\mathbf{z}_t$ given $\mathbf{z}_{t-1}$, $\mathbf{y}_{t-1}$ and $\mathbf{c}$ (Eq. (6.6))
4. Compute $p(y_t|y_{<t}, X)$ (Eq. (6.7))
5. Sample $\tilde{y}_t$ from the compute distribution
6. Repeat (3)–(5) until $\tilde{y}_t = \langle\text{eos}\rangle$

After taking these steps, we get a sample $\tilde{Y} = \left(\tilde{y}_1, \ldots, \tilde{y}_{|\tilde{Y}|}\right)$ given a source sentence X. Of course, there is no guarantee that this will be a good translation of X. In order to find a good translation, meaning a translation with a high probability $P(\tilde{Y}|X)$, we need to repeatedly sample multiple translations from $P(Y|X)$ and choose one with the high probability.

This is not too desirable, as it is not clear how many translations we need to sample from $P(Y|X)$ and also it will likely be computationally expensive. We must wonder whether we can solve the following optimization problem directly:

$$\tilde{Y} = \arg\max_Y \log P(Y|X).$$

Unfortunately, the exact solution to this requires evaluating $P(Y|X)$ for every possible Y. Even if we limit our search space of Y to consist of only sentences of length up to a finite number, it will likely become too large (the cardinality of the set grows exponentially with respect to the number of words in a translation.) Thus, it only makes sense to solving the optimization problem above approximately.

Approximate Decoding: Beamsearch Although it is quite clear that finding a translation $\tilde{Y}$ that maximizes the log-probability $\log P(\tilde{Y}|X)$ is extremely expensive, we will regardlessly try it here.

One very natural way to enumerate all possible target sentences and simultaneously computing the log-probability of each and every one of them is to start from all possible first word, compute the probabilities of them, and from each potential first word branch into all possible second words, and so on. This procedure forms a tree, and any path from the root of this tree to any intermediate node is a valid, but perhaps very unlikely, sentence. See Fig. 6.4 for the illustration. The conditional probabilities of all these paths, or sentences, can be computed as we expand this tree down by simply following Eq. (6.2).

Of course, we cannot compute the conditional probabilities of all possible sentences. Hence, we must resort to some kind of *approximate search*. Wait, *search*? Yes, this whole procedure of finding the most likely translation is equivalent to searching through a space, in this case a tree, of all possible sentences for one sentence that has the highest conditional probability.

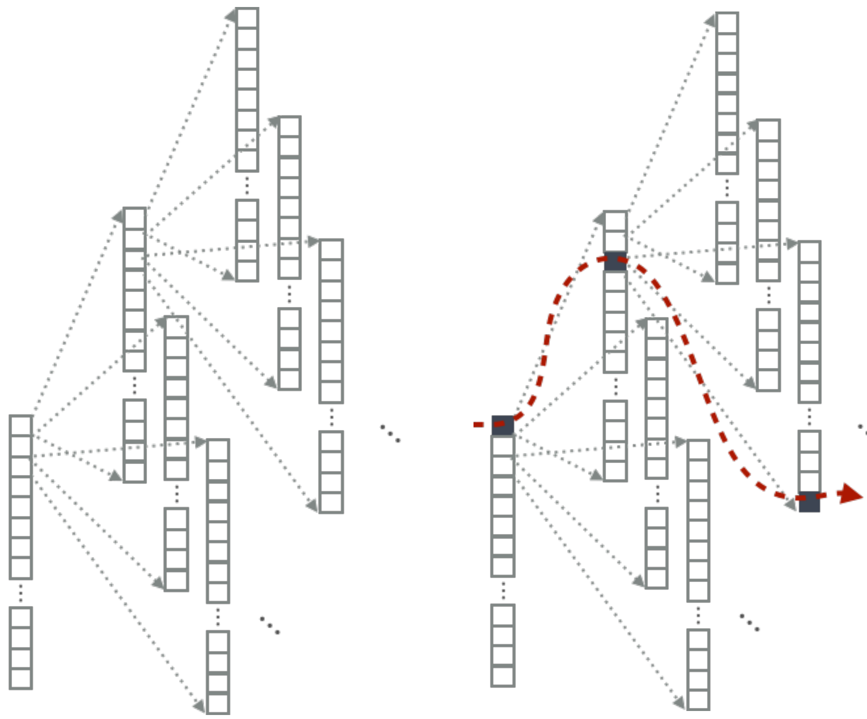


Figure 6.4: (a) Search space depicted as a tree. (b) Greedy search.

The most basic approach to approximately searching for the most likely translation is to choose only a single branch at each time step t. In other words,

$$\hat{y}_t = \arg\max_{w' \in V} \log p(y_t = w' | \hat{y}_{<t}, X),$$

where the conditional probability is defined in Eq. (6.7), and $\hat{y}_{<t} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{t-1})$ is a sequence of greedily-selected target words up to the $(t-1)$-th step. This procedure is repeated until the selected $\hat{y}_t$ is a symbol corresponding to the end of the translation (often denoted as $\langle \text{eos} \rangle$.) See Fig. 6.4 (b) for illustration.

There is a big problem of this *greedy search*. That is, as soon as it makes one mistake at one time step, there is no way for this search procedure to recover from this mistake. This happens because the conditional distributions at later steps depend on the choices made earlier.

Consider the following two sequences: (w_1, w_2) and (w'_1, w_2). These sequences' probabilities are

$$p(w_1, w_2) = p(w_1)p(w_2|w_1),$$
$$p(w'_1, w_2) = p(w'_1)p(w_2|w'_1)$$

Let's assume that

$$\lambda p(w_1) = p(w'_1),$$

where $0 < \lambda < 1$, meaning that $p(w_1) > p(w'_1)$. In this case, the greedy search will choose w_1 over w'_1 and ignore w'_1.

Now we can see that there's a problem with this. Let's assume that

$$\lambda p(w_2|w_1) < p(w_2|w'_1) \iff p(w_2|w_1) < \frac{1}{\lambda} p(w_2|w'_1)$$

where λ was defined earlier. In this case,

$$\begin{aligned} p(w_1, w_2) =& p(w_1)p(w_2|w_1) = \lambda p(w'_1)p(w_2|w_1) \\ <& \cancel{\lambda} p(w'_1) \frac{1}{\cancel{\lambda}} p(w_2|w'_1) = p(w'_1)p(w_2|w'_1) = p(w'_1, w_2). \end{aligned}$$

In short,

$$p(w_1, w_2) < p(w'_1, w_2).$$

It means that the sequence (w'_1, w_2) is more likely than (w_1, w_2), but the greedy search algorithm is unable to notice this, because simply $p(w_1) > p(w'_1)$.

Unfortunately, the only way to completely avoid this undesirable situation is to consider all the possible paths starting from the very first time step. This is exactly the reason why we introduced the greedy search in the first place, but the greedy search is *too* greedy. The question is then whether there is something in between the exact search and the greedy search.

Beam Search Let us start from the very first position $t = 1$. First, we compute the conditional probabilities of all the words in the vocabulary:

$$p(y_1 = w|X) \text{ for all } w \in V.$$

Among these, we choose the K most likely words and initialize the K hypotheses:

$$(w^1_1), (w^1_2), \ldots, (w^1_K)$$

We use the subscript to denote the hypothesis and the subscript the time step. As an example, w^1_1 is the first hypothesis at time step 1.

For each hypothesis, we compute the next conditional probabilities of all the words in the vocabulary:

$$p(y_2 = w|y_{<1} = (w^1_i), X) \text{ for all } w \in V,$$

where $i = 1, \ldots, K$. We then have $K \times |V|$ candidates with the corresponding probabilities:

$$K \left\{ \underbrace{\begin{array}{ccc} p(w^1_1, w^2_{c,1}), & \ldots, & p(w^1_1, w^2_{c,|V|}) \\ p(w^1_2, w^2_{c,1}), & \ldots, & p(w^1_2, w^2_{c,|V|}) \\ & \vdots & \\ p(w^1_K, w^2_{c,1}), & \ldots, & p(w^1_K, w^2_{c,|V|}) \end{array}}_{|V|} \right.$$

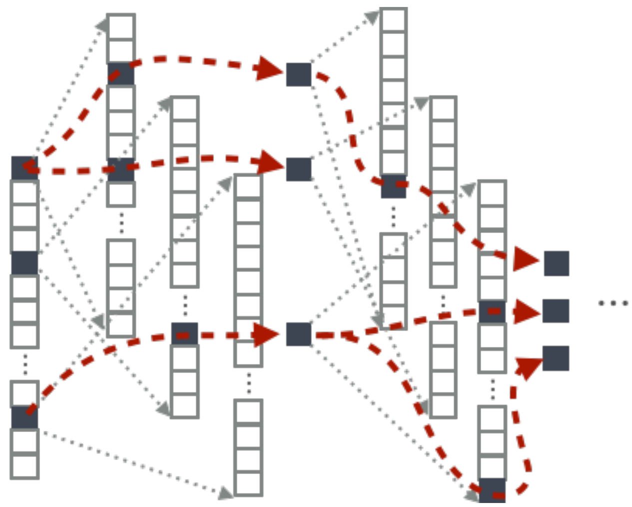


Figure 6.5: Beam search with the beam width set to 3.

Among these $K \times |V|$ candidates, we choose the K most likely candidates:

$$(w_1^1, w_1^2), (w_2^1, w_2^2), \ldots, (w_K^1, w_K^2).$$

Starting from these K new hypotheses, we repeat the process of computing the probabilities of all $K \times |V|$ possible candidates and choosing among them the K most likely new hypotheses.

It should be clear that this procedure, called *beam search* and shown in Fig. 6.5, becomes equivalent to the exact search, as $K \to \infty$. Also, when $K = 1$, this procedure is equivalent to the greedy search. In other words, this beam search interpolates between the exact search, which is computationally intractable but exact, and the greedy search, which is computationally very cheap but probably quite inexact, by changing the size K of hypotheses maintained throughout the search procedure.

How do we choose K? One might mistakenly think that we can simply use as large K as possible given the constraints on computation and memory. Unfortunately, this is not necessarily true, as this interpolation by K is not monotonic. That is, the quality of the translation found by the beam search with a larger K is not necessarily better than the translation found with a smaller K.

Let us consider the case of vocabulary having three symbols $\{a, b, c\}$ and any valid translation being of a length 3. In the first step, we have

$$p(a) = 0.5, p(b) = 0.15, p(c) = 0.45.$$

In the case of $K = 1$, i.e., *greedy search*, we choose a. If $K = 2$, we will keep (a) and (c).

Given a as the first symbol, we have

$$p(a|a) = 0.4, p(b|a) = 0.3, p(c|a) = 0.3,$$

in which case, we keep (a,a) with $K=1$. With $K=2$, we should check also

$$p(a|c)=0.45, p(b|c)=0.45, p(c|c)=0.1,$$

from which we maintain the hypotheses (c,a) and (c,b) (0.45×0.45 and 0.45×0.45, respectively.) Note that with $K=2$, we have discarded (a,a).

Now, the greedy search ends by computing the last conditional probabilities:

$$p(a|a,a)=0.9, p(b|a,a)=0.05, p(c|a,a)=0.05.$$

The final verdict from the greedy search is therefore (a,a,a) with its probability being $0.5 \times 0.4 \times 0.9 = 0.18$.

What happens with the beam search having $K=2$? We need to check the following conditional probabilities:

$$\begin{aligned} p(a|c,a)=0.7, p(b|c,a)=0.2, p(c|c,a)=0.1 \\ p(a|c,b)=0.4, p(b|c,b)=0.0, p(c|c,b)=0.6 \end{aligned}$$

From here we consider (c,a,a) and (c,b,c) with the corresponding probabilities $0.45 \times 0.45 \times 0.7 = 0.14175$ and $0.45 \times 0.45 \times 0.6 = 0.1215$. Among these two, (c,a,a) is finally chosen, due to its higher probability than that of (c,b,c).

In summary, the greedy search found (a,a,a) whose probability is

$$p(a,a,a)=0.18,$$

and the beam search with $K=2$ found (c,a,a) whose probability is

$$p(c,a,a)=0.14175.$$

Even with a larger K, the beam search found a worse translation!

Now, clearly, what one can do is to set the maximum beam width $\bar{K}$ and try with all possible $1 \leq K \leq \bar{K}$. Among the translations given by $\bar{K}$ beam search procedures, the best translation can be selected based on their corresponding probabilities. From the point of view of computational complexity, this is perhaps the best approach to upper-bound the worst-case memory consumption. Doing the beam search once with $\bar{K}$ or multiple beam searches with $K=1,\ldots,\bar{K}$ are equivalent in terms of memory consumption, i.e., both are $O(K|V|)$. Furthermore, the worst-case computation is $O(K|V|)$ (assuming a constant time computation for computing each conditional probability.) In practice however, the constant in front of $K|V|$ does matter, and we often choose K based on the translation quality of the validation set, after trying a number of values–$\{1,2,4,8,16\}$.

If you're interested in how to improve beam search by backtracking so that the beam search becomes *complete*, refer to, e.g., [39, 94]. If you're interested in general search strategies, refer to [79]. Also, in the context of statistical machine translation, it is useful to read [54].

6.3 Attention-based Neural Machine Translation

One important property of the simple encoder-decoder model for neural machine translation (from Sec. 6.2) is that a whole source sentence is compressed into a single real-valued vector **c**. This sounds okay, since the space of all possible source sentences is *countable*, while the context vector space $[-1,1]^d$ is *uncountable*. There exists a mapping from this sentence space to the context vector space, and all we need to ensure is that training the simple encoder-decoder model finds this mapping. This is conditioned on the assumption that the hypothesis space[9] defined by the model architecture–the number of hidden units and parameters– includes this mapping from any source sentence to a context vector.

Unfortunately, considering the complexity of any natural language sentence, it is quite easy to guess that this mapping must be highly nonlinear and will require a huge encoder, and consequently, a huge decoder to map back from a context vector to a target sentence. In fact, this fact was empirically validated last year (2014), when the almost identical models from two groups [87, 24] showed vastly different performances on the same English–French translation task. The only difference there was that the authors of [87] used a *much* larger model than the authors of [24] did.

At a more fundamental level there's a question of whether a natural language sentence *should* be fully represented as a single vector. For instance, there is now a famous quote by Prof. Raymond Mooney[10] of the University of Texas at Austin: "You can't cram the meaning of a whole %&!$# sentence into a single $&!#* vector!"[11] Though, our goal is not in answering this fundamental question from linguistics.

Our goal is rather to investigate the possibility of avoiding this situation of having to learn a highly nonlinear, complex mapping from a source sentence to a single vector. The question we are more interested in is whether *there exists a neural network that can handle a variable-length sentence by building a variable-length representation of it*. Especially, we are interested in whether we can build a neural machine translation system that can exploit a variable-length context representation.

Variable-length Context Representation In the simple encoder-decoder model, a source sentence, regardless of its length, was mapped to a single context vector by a recurrent neural network:

$$\mathbf{h}_t = \phi_{\text{enc}}\left(\mathbf{h}_{t-1}, \mathbf{E}_x^\top \mathbf{x}_t\right).$$

See Eq. (6.5) and the surrounding text for more details.

Instead, here we will encode a source sentence $X = (x_1, x_2, \ldots, x_{T_x})$ with a *set C of context vectors* $\mathbf{h}_t$'s. This is achieved by having two recurrent neural networks rather than a single recurrent neural networks, as in the simple encoder-decoder model. The first recurrent neural network, to which we will refer as a *forward recurrent neural network*, reads the source sentence as usual and results in a set of *forward memory*

[9] See Sec. 2.3.2.

[10] https://www.cs.utexas.edu/~mooney/

[11] http://nlpers.blogspot.com/2014/09/amr-not-semantics-but-close-maybe.html

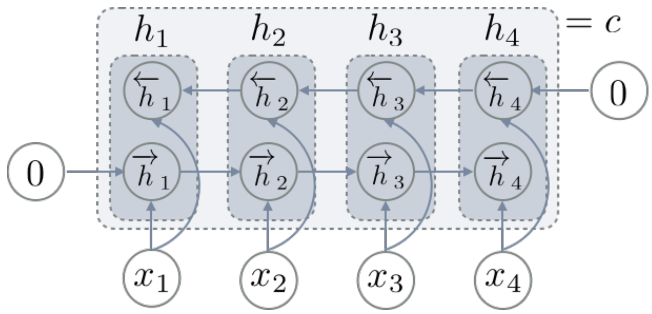


Figure 6.6: An encoder with a bidirectional recurrent neural network

states $\overrightarrow{\mathbf{h}}_t$, for $t = 1, \ldots, T_x$. The second recurrent neural network, a *reverse recurrent neural network*, reads the source sentence in a reverse order, starting from x_{T_x} to x_1. This reverse network will output a sequence of *reverse memory states* $\overleftarrow{\mathbf{h}}_t$, for $t = 1, \ldots, T_x$.

For each x_t, we will concatenate $\overrightarrow{\mathbf{h}}_t$ and $\overleftarrow{\mathbf{h}}_t$ to form a *context-dependent vector* $\mathbf{h}_t$:

$$\mathbf{h}_t = \begin{bmatrix} \overrightarrow{\mathbf{h}}_t \\ \overleftarrow{\mathbf{h}}_t \end{bmatrix} \tag{6.9}$$

We will form a *context set* with these context-dependent vectors $c = \{\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_{T_x}\}$. See Fig. 6.6 for the graphical illustration of this process.

Now, why is $\mathbf{h}_t$ a *context-dependent vector*? We should look at what the input was to a function that computed $\mathbf{h}_t$. The first half of $\mathbf{h}_t$, $\overrightarrow{\mathbf{h}}_t$, was computed by

$$\overrightarrow{\mathbf{h}}_t = \phi_{\text{fenc}}\left(\phi_{\text{fenc}}\left(\cdots, \mathbf{E}_x^\top \mathbf{x}_{t-1}\right), \mathbf{E}_x^\top \mathbf{x}_t\right),$$

where ϕ_{fenc} is a forward recurrent activation function. From this we see that $\overrightarrow{\mathbf{h}}_t$ was computed by all the source words up to t, i.e., $\mathbf{x}_{\leq t}$. Similarly,

$$\overleftarrow{\mathbf{h}}_t = \phi_{\text{renc}}\left(\phi_{\text{renc}}\left(\cdots, \mathbf{E}_x^\top \mathbf{x}_{t+1}\right), \mathbf{E}_x^\top \mathbf{x}_t\right),$$

where ϕ_{renc} is a reverse recurrent activation function, and $\overleftarrow{\mathbf{h}}_t$ depends on all the source words from t to the end, i.e., $\mathbf{x}_{\geq t}$.

In summary, $\mathbf{h}_t = \left[\overrightarrow{\mathbf{h}}_t^\top; \overleftarrow{\mathbf{h}}_t^\top\right]^\top$ is a vector representation of the t-th word, x_t, with respect to all the other words in the source sentence. This is why $\mathbf{h}_t$ is a context-dependent representation. But, then, what is the difference among all those context-dependent representations $\{\mathbf{h}_1, \ldots, \mathbf{h}_{T_x}\}$? We will discuss this shortly.

Decoder with Attention Mechanism Before anything let us think of what the memory state $\mathbf{z}_t$ of the decoder (from Eq. (6.6)) does:

$$\mathbf{z}_t = \phi_{\text{dec}}\left(\phi_{\text{dec}}\left(\phi_{\text{dec}}\left(\cdots, \left[\mathbf{E}_y^\top \mathbf{y}_{t-3}; \mathbf{c}\right]\right), \left[\mathbf{E}_y^\top \mathbf{y}_{t-2}; \mathbf{c}\right]\right) \left[\mathbf{E}_y^\top \mathbf{y}_{t-1}; \mathbf{c}\right]\right)$$

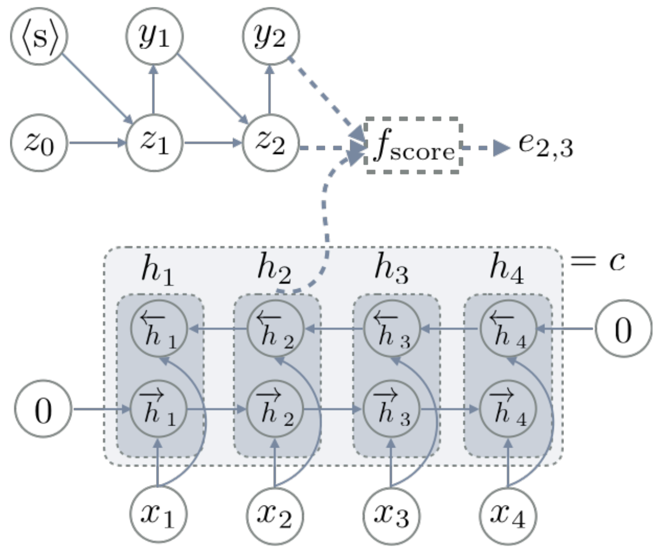


Figure 6.7: Illustration of how the relevance score $e_{2,3}$ of the second context vector $\mathbf{h}_2$ at time step 3 (dashed curves and box.)

It is computed based on all the generated target words so far $(\tilde{y}_1, \tilde{y}_2, \ldots, \tilde{y}_{t-1})$ and the context vector[12] $\mathbf{c}$ which is the summary of the source sentence. The very reason why I designed the decoder in this way is so that the memory state $\mathbf{z}_t$ is informative of which target word should be generated at time t *after* generating the first $t-1$ target words *given* the source sentence. In order to do so, $\mathbf{z}_t$ must encode *what have been translated so far* among the words that are supposed to be translated (which is encoded in the context vector $\mathbf{c}$.) Let's keep this in mind.

In order to compute the new memory state $\mathbf{z}_t$ with a *context set* $C = \{\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_{T_x}\}$, we must first get one vector out of T_x context vectors. Why is this necessary? Because we cannot have an infinitely large number of parameters to cope with any number of context vectors. Then, how can we get a single vector from an unspecified number of context vectors $\mathbf{h}_t$'s?

First, let us score each context vector $\mathbf{h}_j$ ($j = 1, \ldots, T_x$) based on how *relevant* it is for translating a next target word. This scoring needs to be based on (1) the previous memory state $\mathbf{z}_{t-1}$ which summarizes what has been translated up to the $(t-2)$-th word[13], (2) the previously generated target word $\tilde{y}_{t-1}$, and (3) the j-th context vector $\mathbf{h}_j$:

$$e_{j,t} = f_{\text{score}}(\mathbf{z}_{t-1}, \mathbf{E}_y^\top \tilde{y}_{t-1}, \mathbf{h}_j). \tag{6.10}$$

Conceptually, the score $e_{j,t}$ will be computed by *comparing* $(\mathbf{z}_{t-1}, \tilde{y}_{t-1})$ with the context vector $\mathbf{c}_j$. See Fig. 6.7 for graphical illustration.

[12] We will shortly switch to using a context *set* instead.

[13] Think of why this is only up to the $(t-2)$-th word not up to the $(t-1)$-th one.

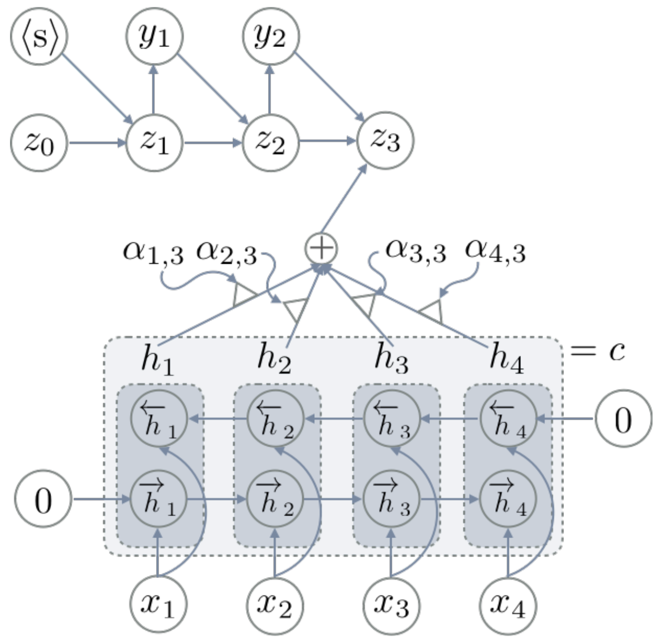


Figure 6.8: Computing the new memory state $\mathbf{z}_t$ of the decoder based on the previous memory state $\mathbf{z}_{t-1}$, the previous target word $\tilde{y}_{t-1}$ and the weighted average of context vectors according to the attention weights.

Once the scores for all the context vectors $\mathbf{h}_j$'s ($j = 1, \ldots, T_x$) are computed by f_{score}, we normalize them with a softmax function:

$$\alpha_{j,t} = \frac{\exp(e_{j,t})}{\sum_{j'=1}^{T_x} \exp(e_{j',t})}. \tag{6.11}$$

We call these normalized scores the *attention weights*, as they correspond to how much the *decoder attends* to each of the context vectors. This whole process of computing the attention weights is often referred to as an *attention mechanism* (see, e.g., [23].)

We take the weighted average of the context vectors with these attention weights:

$$\mathbf{c}_t = \sum_{j=1}^{T_x} \alpha_{j,t} \mathbf{h}_j \tag{6.12}$$

This weighted average is used to compute the new memory state $\mathbf{z}_t$ of the decoder, which is identical to the decoder's update equation from the simple encoder-decoder model (see Eq. (6.6)) except that $\mathbf{c}_t$ is used instead of $\mathbf{c}$ ((a) in the equation below):

$$\mathbf{z}_t = \phi_{\text{dec}} \left(\mathbf{z}_{t-1}, \left[\mathbf{E}_y^\top \mathbf{y}_{t-1}; \underbrace{\mathbf{c}_t}_{\text{(a)}} \right] \right)$$

See Fig. 6.8 for the graphical illustration of how it works.

Given the new memory state $\mathbf{z}_t$ of the decoder, the output probabilities of all the target words in a vocabulary happen without any change from the simple encoder-decoder model in Sec. 6.2.

We will call this model, which has a bidirectional recurrent neural network as an encoder and a decoder with the attention mechanism, an *attention-based encoder-decoder model*. This approach was proposed last year (2014) in the context of machine translation in [1] and has been studied extensively in [66].

6.3.1 What does the Attention Mechanism do?

One important thing to notice is that this attention-based encoder-decoder model can be reduced to the simple encoder-decoder model easily. This happens when the attention mechanism f_{score} in Eq. (6.10) returns a constant regardless of its input. When this happens, the context vector $\mathbf{c}_t$ at each time step t (see Eq. (6.12)) is same for all the time steps $t = 1, \ldots, T_y$:

$$\mathbf{c}_t = \frac{1}{T_x} \sum_{j=1}^{T_x} \mathbf{h}_j.$$

The encoder effectively maps the whole input sentence into a single vector, which was at the core of the simple encoder-decoder model from Sec. 6.2.

This is not the only situation in which this type of behaviour happens. Another possible scenario is for the encoder to make the last memory states, $\overrightarrow{\mathbf{h}}_{T_x}$ and $\overleftarrow{\mathbf{h}}_1$, of the forward and reverse recurrent neural networks to have a special mark telling that these are the last states. The attention mechanism then can exploit this to assign a large score to these two memory states (but still constant across time t.) This will become even closer to the simple encoder-decoder model.

The question is how we can avoid these degenerate cases. Or, *is it necessary for us to explicitly make these degenerate cases unlikely?* Of course, there is no single answer to this question. Let me give you my answer, which may differ from others' answer: *no*.

The goal of introducing a novel network architecture is to guide a model according to our intuition or scientific observation so that it will do a better job at a target task. In our case, the attention mechanism was introduced based on our observation, and some intuition, that it is not desirable to ask the encoder to compress a whole source sentence into a single vector.

This incorporation of *prior knowledge* however should not put a *hard constraint*. We give a model a possibility of exploiting this prior knowledge, but should not force the model to use this prior knowledge exclusively. As this prior knowledge, based on our observation of a small portion of data, is not likely to be true in general, the model must be able to ignore this, if the data does not exhibit the underlying structure corresponding to this prior knowledge. In this case of attention-based encoder-decoder model, the existence of those degenerate cases above is a direct evidence of what this attention-based model can do, if there is no such underlying structure present in the data.

Then, a natural next question is whether there are such structures that can be well exploited by this attention mechanism in real data. If we train this attention-based encoder-decoder model on the parallel corpora we discussed earlier in Sec. 6.1.1, what kind of structure does this attention-based model learn?

In order to answer this question, we must first realize that we can easily visualize what is happening inside this attention-based model. First, note that given a pair of source X and target Y sentences,[14] the attention-based model computes an *alignment* matrix $A \in [0,1]^{|X|\times|Y|}$:

$$A = \begin{bmatrix} \alpha_{1,1} & \alpha_{1,2} & \cdots & \alpha_{1,|Y|} \\ \alpha_{2,1} & \alpha_{2,2} & \cdots & \alpha_{2,|Y|} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{|X|,1} & \alpha_{|X|,2} & \cdots & \alpha_{|X|,|Y|} \end{bmatrix},$$

where $\alpha_{j,t}$ is defined in Eq. (6.11).

Each column $\mathbf{a}_t$ of this alignment matrix A is how well each source word (based on its *context-dependent vector representation* from Eq. (6.9)) is aligned to the t-th target word. Each row $\mathbf{b}_j$ similarly shows how well each target word is aligned to the content-dependent vector of the j-th source word. In other words, we can simply draw the alignment matrix A as if it were a gray scale 2-D image.

In Fig. 6.9, the visualization of four alignment matrices is presented. It is quite clear, especially to a French-English bilingual speaker, that the model indeed captured the underlying structure of word/phrase mapping between two languages. For instance, focus on "European Economic Area" in Fig. 6.9 (a). The model correctly noticed that "Area" corresponds to "zone", "Economic" to "économique", and "European" to "européenne", without any supervision about this type of alignment.

This is nice to see that the model was able to notice these regularities from data without any explicit supervision. However, the goal of introducing the attention mechanism was not to get these pretty figures. After all, our goal is not to build an interpretable model, but a model that is predictive of the correct output given an input (see Chapter 1 and [14].) In this regard, how much does the introduction of the attention mechanism help?

In [1], the attention-based encoder-decoder model was compared against the simple encoder-decoder model in the task of English-French translation. They observed the relative improvement of up to 60% (in terms of BLEU, see Sec. 6.1.2,) as shown in Table 6.1. Furthermore, by using some of the latest techniques, such as handling large vocabularies [49], building a vocabulary of subword units [82] and variants of the attention mechanism [66], it has been found possible to achieve a better translation quality with neural machine translation than the existing state-of-the-art translation systems.

[14] Note that if you're given only a source sentence, you can let the model translate and align simultaneously.

Model	BLEU	Rel. Improvement
Simple Enc–Dec	17.82	–
Attention-based Enc–Dec	28.45	+59.7%
Attention-based Enc–Dec (LV)	34.11	+90.7%
Attention-based Enc–Dec (LV)*	**37.19**	**+106.0%**
State-of-the-art SMT$^\circ$	37.03	–

Table 6.1: The translation performances and the relative improvements over the simple encoder-decoder model on an English-to-French translation task (WMT'14), measured by BLEU [1, 49]. $\star$: an ensemble of multiple attention-based models. $\circ$: the state-of-the-art phrase-based statistical machine translation system [35].

6.4 Warren Weaver's Memorandum

In 1949 Warren Weaver[15] wrote a memorandum, titled $\langle$Translation$\rangle$ on machine translation [91]. Although this text was written way before computers have become ubiquitous,[16] there are many interesting ideas that are closely related to what we have discussed so far in this chapter. Let us go over some parts of the Weaver's memorandum and see how the ideas there corresponds to modern-day machine translation.

Necessity of Linguistic Knowledge Weaver talks about a distinguished mathematician P who was surprised by his colleague. His colleague "had an amateur interest in cryptography", and one day presented P his method to "decipher" an encrypted Turkish text successfully. "The most important point", according to Weaver, from this instance is that "the decoding was done by someone who did not know Turkish." Now, this sounds familiar, doesn't it?

As long as there was a parallel corpus, we are able to use neural machine translation models, described throughout this chapter, without ever caring about which languages we are training a model to translate between. Especially if we decide to consider each sentence as a sequence of *characters*,[17] there is almost no need for any linguistic knowledge when building these neural machine translation systems.

This lack of necessity for linguistic knowledge is not new. In fact, the most widely studied and used machine translation approach, which is (count-based) statistical machine translation [17, 56], does not require any prior knowledge about source and target languages. All it needs is a large corpus.

Importance of Context Recall from Sec. 6.3 that the encoder of an attention-based neural machine translation uses a *bidirectional* recurrent neural network in order to obtain a context set. Each vector in the context set was considered a *context-dependent*

[15] Yes, this is the very same Weaver after which the building of the Courant Institute of Mathematical Sciences has been named.

[16] Although Weaver talks about modern computers over and over in his memorandum, what he refers to is not exactly what we think of computers as these days.

[17] In fact, only very recently people have started investigating the possibility of building a machine translation system based on character sequences [63]. This has been made possible due to the recent success of neural machine translation.

vector, as it represents what the center word means with respect to all the surrounding words. This context dependency is a necessary component in making the whole attention-based neural machine translation, as it helps disambiguating the meaning of each word and also distinguishing multiple occurrences of a single word by their context.

Weaver discusses this extensively in Sec. 3–4 in his memorandum. First, to Weaver, it was "amply clear that a translation procedure that does little more than handle a one-to-one correspondence of words can not hope to be useful .. in which the problems of .. multiple meanings .. are frequent." In other words, it is simply not possible to look at each word separately from surrounding words (or context) and translate it to a corresponding target word, because there is uncertainty in the meaning of the source word which can only be resolved by taking into account its context.

So, what does Weaver propose in order to address this issue? He proposes in Sec. 5 that if "one can see not only the central word in question, but also say N words on either side, then if [sic] N is large enough one can unambiguously decide the meaning of the central word." If we consider only a single sentence and take the infinite limit of $N \to \infty$, we see that what Weaver refers to is exactly the bidirectional recurrent neural network used by the encoder of the attention-based translation system. Furthermore, we see that the continuous bag-of-words language model, or Markov random field based language model, from Sec. 5.4.2 exactly does what Weaver proposed by setting N to a finite number.

In Sec. 5.2.1, we talked about the issue of data sparsity, and how it is desirable to have a larger N but it's often not a good idea statistically to do so. Weaver was also worried about this by saying that "it would hardly be practical to do this by means of a generalized dictionary which contains all possible phases [sic] $2N+1$ words long; for the number of such phases [sic] is horrifying." We learned that this issue of data sparsity can be largely avoided by adopting a fully parametric approach instead of a table-based approach in Sec. 5.4.

Common base of human communications Weaver suggested in the last section of his memorandum that "perhaps the way" for translation "is to descend, from each language, down to the common base of human communication – the real but as yet undiscovered universal language – and then re-emerge by whatever particular route is convenient." He specifically talked about a "universal language", and this makes me wonder if we can consider the memory state of the recurrent neural networks (both of the encoder and decoder) as this kind of intermediate language. This intermediate language radically departs from our common notion of natural languages. Unlike conventional languages, it does not use discrete symbols, but uses continuous vectors. This use of continuous vectors allows us to use simple arithmetics to manipulate the meaning, as well as its surface realization.[18]

This view may sound radical, considering that what we've discussed so far has been confined to translating from one language to another. After all, this universal language

[18] If you find this view too radical or fascinating, I suggest you to look at the presentation slides by Geoff Hinton at `https://drive.google.com/file/d/0B16RwCMQqrtdMWFaeThBTC1mZkk/view?usp=sharing`

of ours is very specific to only a single source language with respect to a single target language. This is however not a constraint on the neural machine translation by design, but simply a consequence of our having focused on this specific case.

Indeed, in this year (2015), researchers have begun to report that it is possible to build a neural machine translation model that considers multiple languages, and even further multiple tasks [34, 65]. More works in this line are expected, and it will be interesting to see if Weaver's prediction again turns out to be true.

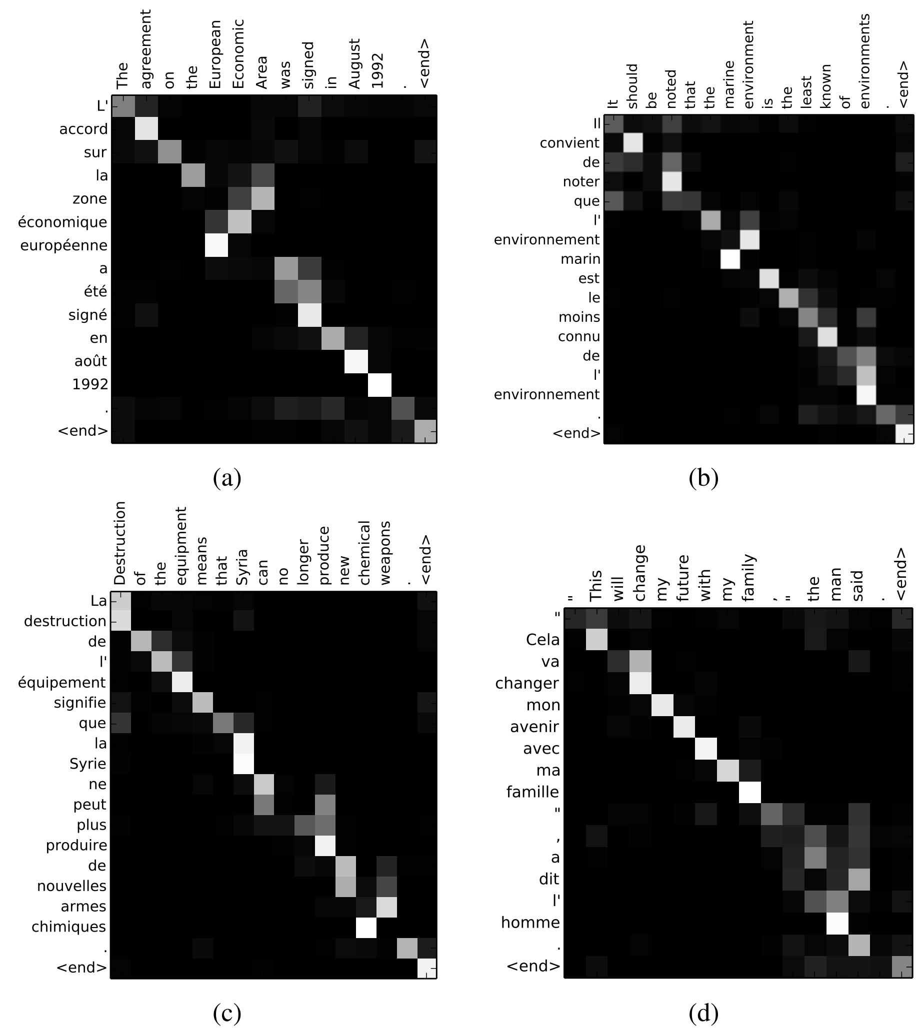


Figure 6.9: Visualizations of the four sample alignment matrices. The alignment matrices were computed from an attention-based translation model trained to translate a sentence in English to French. Reprinted from [1].

Chapter 7

Final Words: Beyond Natural Languages

7.1 Multimedia Description Generation as Translation

TBD

7.2 Natural Language Understanding with Word Knowledge

TBD

7.3 Summary

TBD

Soon to appear

Bibliography

[1] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[2] P. Baltescu and P. Blunsom. Pragmatic neural language modelling in machine translation. *arXiv preprint arXiv:1412.7119*, 2014.

[3] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*, 2012.

[4] A. G. Baydin, B. A. Pearlmutter, and A. A. Radul. Automatic differentiation in machine learning: a survey. *arXiv preprint arXiv:1502.05767*, 2015.

[5] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu. Advances in optimizing recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8624–8628. IEEE, 2013.

[6] Y. Bengio, N. Léonard, and A. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[7] Y. Bengio, H. Schwenk, J.-S. Senécal, F. Morin, and J.-L. Gauvain. Neural probabilistic language models. In *Innovations in Machine Learning*, pages 137–186. Springer Berlin Heidelberg, 2006.

[8] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166, 1994.

[9] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, page 3. Austin, TX, 2010.

[10] J. Besag. Statistical analysis of non-lattice data. *The statistician*, pages 179–195, 1975.

[11] C. M. Bishop. Mixture density networks. 1994.

[12] C. M. Bishop. *Pattern recognition and machine learning*. springer, 2006.

[13] L. Bottou. Online algorithms and stochastic approximations. In D. Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998.

[14] L. Breiman et al. Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical Science*, 16(3):199–231, 2001.

[15] J. S. Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 211–217. Morgan-Kaufmann, 1990.

[16] E. Brochu, V. M. Cora, and N. de Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv:*`1012.2599 [cs.LG]`, Dec. 2010.

[17] P. F. Brown, J. Cocke, S. A. D. Pietra, V. J. D. Pietra, F. Jelinek, J. D. Lafferty, R. L. Mercer, and P. S. Roossin. A statistical approach to machine translation. *Computational linguistics*, 16(2):79–85, 1990.

[18] C. Callison-Burch, M. Osborne, and P. Koehn. Re-evaluation the role of bleu in machine translation research. In *EACL*, volume 6, pages 249–256, 2006.

[19] A. Carnie. *Syntax: A generative introduction*. John Wiley & Sons, 2013.

[20] M. Cettolo, C. Girardi, and M. Federico. Wit^3: Web inventory of transcribed and translated talks. In *Proceedings of the 16th Conference of the European Association for Machine Translation (EAMT)*, pages 261–268, Trento, Italy, May 2012.

[21] O. Chapelle, B. Schölkopf, and A. Zien, editors. *Semi-Supervised Learning*. MIT Press, Cambridge, MA, 2006.

[22] S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 310–318. Association for Computational Linguistics, 1996.

[23] K. Cho, A. Courville, and Y. Bengio. Describing multimedia content using attention-based encoder–decoder networks. 2015.

[24] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[25] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[26] K. Cho, B. van Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014)*, Oct. 2014.

[27] N. Chomsky. A review of B. F. skinner's verbal behavior. *Language*, 35(1):26–58, 1959.

[28] N. Chomsky. Linguistic contributions to the study of mind (future). *Language and thinking*, pages 323–364, 1968.

[29] N. Chomsky. *Syntactic structures*. Walter de Gruyter, 2002.

[30] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 12:2493–2537, 2011.

[31] T. M. Cover. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Transactions on Electronic Computers*, EC-14(3):326–334, 1965.

[32] J. Denker and Y. Lecun. Transforming neural-net output levels to probability distributions. In *Advances in Neural Information Processing Systems 3*. Citeseer, 1991.

[33] M. Denkowski and A. Lavie. Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the EACL 2014 Workshop on Statistical Machine Translation*, 2014.

[34] D. Dong, H. Wu, W. He, D. Yu, and H. Wang. Multi-task learning for multiple language translation. ACL, 2015.

[35] N. Durrani, B. Haddow, P. Koehn, and K. Heafield. Edinburgh's phrase-based machine translation systems for WMT-14. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 97–104. Association for Computational Linguistics Baltimore, MD, USA, 2014.

[36] J. R. Firth. A synopsis of linguistic theory 1930-1955. *Oxford: Philological Society*, 1957.

[37] R. Fletcher. *Practical Methods of Optimization*. Wiley-Interscience, New York, NY, USA, 2nd edition, 1987.

[38] M. L. Forcada and R. P. Ñeco. Recursive hetero-associative memories for translation. In *Biological and Artificial Computation: From Neuroscience to Technology*, pages 453–462. Springer, 1997.

[39] D. Furcy and S. Koenig. Limited discrepancy beam search. In *IJCAI*, pages 125–131, 2005.

[40] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[41] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[42] I. Goodfellow, D. Warde-farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1319–1327, 2013.

[43] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber. Lstm: A search space odyssey. *arXiv preprint arXiv:1503.04069*, 2015.

[44] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv preprint arXiv:1502.01852*, 2015.

[45] K. Heafield, I. Pouzyrevsky, J. H. Clark, and P. Koehn. Scalable modified Kneser-Ney language model estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 690–696, Sofia, Bulgaria, August 2013.

[46] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies*, volume 1. A field guide to dynamical recurrent neural networks. IEEE Press, 2001.

[47] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[48] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew. Extreme learning machine: Theory and applications. *Neurocomputing*, 70(1—3):489–501, 2006.

[49] S. Jean, K. Cho, R. Memisevic, and Y. Bengio. On using very large target vocabulary for neural machine translation. In *ACL 2015*, 2014.

[50] Y. Jernite, A. M. Rush, and D. Sontag. A fast variational approach for learning markov random field language models. *32nd International Conference on Machine Learning (ICML)*, 2015.

[51] R. Jozefowicz, W. Zaremba, and I. Sutskever. An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 2342–2350, 2015.

[52] Y. Kim, Y. Jernite, D. Sontag, and A. M. Rush. Character-aware neural language models. *arXiv preprint arXiv:1508.06615*, 2015.

[53] R. Kneser and H. Ney. Improved backing-off for m-gram language modeling. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 1, pages 181–184. IEEE, 1995.

[54] P. Koehn. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In *Machine translation: From real users to research*, pages 115–124. Springer, 2004.

[55] P. Koehn. Europarl: A parallel corpus for statistical machine translation. In *MT summit*, volume 5, pages 79–86. Citeseer, 2005.

[56] P. Koehn, F. J. Och, and D. Marcu. Statistical phrase-based translation. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 48–54. Association for Computational Linguistics, 2003.

[57] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[58] T. S. Kuhn. *The structure of scientific revolutions*. University of Chicago press, 2012.

[59] Q. V. Le, N. Jaitly, and G. E. Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*, 2015.

[60] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[61] Y. LeCun, L. Bottou, G. Orr, and K. R. Müller. Efficient BackProp. In G. Orr and K. Müller, editors, *Neural Networks: Tricks of the Trade*, volume 1524 of *Lecture Notes in Computer Science*, pages 5–50. Springer Verlag, 1998.

[62] O. Levy and Y. Goldberg. Neural word embedding as implicit matrix factorization. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2177–2185. Curran Associates, Inc., 2014.

[63] W. Ling, I. Trancoso, C. Dyer, and A. W. Black. Character-based neural machine translation. *arXiv preprint arXiv:1511.04586*, 2015.

[64] D. G. Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.

[65] M.-T. Luong, Q. V. Le, I. Sutskever, O. Vinyals, and L. Kaiser. Multi-task sequence to sequence learning. *arXiv preprint arXiv:1511.06114*, 2015.

[66] M.-T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[67] C. D. Manning and H. Schütze. *Foundations of statistical natural language processing*. MIT press, 1999.

[68] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[69] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur. Recurrent neural network based language model. In *INTERSPEECH 2010*, pages 1045–1048, 2010.

[70] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[71] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[72] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of The 30th International Conference on Machine Learning*, pages 1310–1318, 2013.

[73] A. Perfors, J. Tenenbaum, and T. Regier. Poverty of the stimulus? a rational approach. In *Annual Conference*, 2006.

[74] K. B. Petersen, M. S. Pedersen, et al. The matrix cookbook. *Technical University of Denmark*, 7:15, 2008.

[75] C. W. Post. *The Three Percent Problem: Rants and Responses on Publishing, Translation, and the Future of Reading*. Open Letter, 2011.

[76] P. Resnik and N. A. Smith. The web as a parallel corpus. *Computational Linguistics*, 29(3):349–380, 2003.

[77] H. Robbins and S. Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[78] F. Rosenblatt. *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Report (Cornell Aeronautical Laboratory). Spartan Books, 1962.

[79] S. Russell and P. Norvig. Artificial intelligence: a modern approach. 1995.

[80] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[81] H. Schwenk. Continuous space language models. *Computer Speech & Language*, 21(3):492–518, 2007.

[82] R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[83] B. F. Skinner. *Verbal behavior*. BF Skinner Foundation, 2014.

[84] J. R. Smith, H. Saint-Amand, M. Plamada, P. Koehn, C. Callison-Burch, and A. Lopez. Dirt cheap web-scale parallel text from the common crawl. In *ACL (1)*, pages 1374–1383, 2013.

[85] M. Snover, B. Dorr, R. Schwartz, L. Micciulla, and J. Makhoul. A study of translation edit rate with targeted human annotation. In *Proceedings of association for machine translation in the Americas*, pages 223–231, 2006.

[86] M. Sundermeyer, H. Ney, and R. Schluter. From feedforward to recurrent lstm neural networks for language modeling. *Audio, Speech, and Language Processing, IEEE/ACM Transactions on*, 23(3):517–529, 2015.

[87] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[88] J. Turian, L. Ratinov, and Y. Bengio. Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics*, pages 384–394. Association for Computational Linguistics, 2010.

[89] L. van der Maaten and G. E. Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, November 2008.

[90] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[91] W. Weaver. Translation. *Machine translation of languages*, 14:15–23, 1955.

[92] T. Winograd. Understanding natural language. *Cognitive psychology*, 3(1):1–191, 1972.

[93] Y. Zhang, K. Wu, J. Gao, and P. Vines. Automatic acquisition of chinese–english parallel corpus from the web. In *Advances in Information Retrieval*, pages 420–431. Springer, 2006.

[94] R. Zhou and E. A. Hansen. Beam-stack search: Integrating backtracking with beam search. In *ICAPS*, pages 90–98, 2005.